引导大语言模型生成计算机可解析内容

Jiaye Wang[1,2*†]
[1] School of Software, South China Normal University
[2] Platform and Content Group, Tencent Inc.
2024-03-26[‡]

摘要

大语言模型 (Large Language Models, LLMs) 能够从大量语料的上下文中学习到模式，其包括词语之间的关系、句子的结构甚至更复杂的语义和语用信息。然而，让预训练语言模型生成结构化、严格遵循约定的内容仍然是一项挑战。

本文提出了一种引导 LLMs 生成计算机高可用内容的方案，无需微调和额外的神经网络推理，通过提前约定的上下文无关文法 (Context-Free Grammar, CFG) 引入基于协程的内容生成约束机制，在自回归模型 Transformer 的解码阶段引导模型采样正确的词元，以构成符合程序约定的形式语言。这将有效地提升 LLMs 生成目标数据结构、类型或指令的稳定性和一致性，降低应用开发和集成的难度。

本文作者先通过“匹配括号对”实验验证了 GPT-2 和 Gemma 等模型在生成 DSL 长度分别大于 36 和 282 时错误率就达到了 95%，说明了当前 LLMs 在特定 DSL 生成上的性能问题。本文作者还提出了基于协程的 DSL 生成框架 YieldLang，并使用 LLMs 在多个任务数据集上进行了实验，包括 JSON、Mermaid 流图和函数调用表达式生成等任务。这些实验表明本文的方法相比基准，其准确率提升到了原来的 109% 到 1160%，并且在最好的情况下能够将 LLMs 生成 JSON 的采样次数降低到基准的约 16.5%，这将有效地提高 LLMs 生成内容对计算机程序的可用性。

关键词：大语言模型，结构化内容生成，计算机辅助编程，约束解码，协程，上下文无关文法

---

[*] 初公开于：https://chinaxiv.org/abs/202403.00340
[†] 作者邮箱：hk-shao@outlook.com
[‡] 版本日期：2024-04-07

Abstract

Large language models (LLMs) have demonstrated remarkable capabilities in learning patterns from massive text corpora, including word relationships, sentence structures, and even complex semantic and pragmatic information. However, it remains challenging to induce pre-trained language models to generate structured content that strictly follows specific conventions.

We propose a scheme for guiding LLMs to generate highly usable content for computers without the need for fine-tuning and additional neural network inference, by introducing coroutine-based content generation constraints through a pre-agreed context-free grammar (CFG), which guides the autoregressive model Transformer to sample the correct tokens during its decoding phase to form a program-compliant form in the decoding phase of the autoregressive model Transformer to form a formal language that conforms to the program conventions. This will effectively improve the stability and consistency of LLMs in generating target data structures, types or instructions, and reduce the difficulty of application development and integration.

We first verified that the error rate of models such as GPT-2 and Gemma reaches 95% when the length of the generated DSLs are greater than 36 and 282, respectively, through the experiment of “matching bracket pairs”, which illustrates the performance problem of some current LLMs in the generation of specific DSLs. We also present YieldLang, a coroutine-based DSL generation framework, and conduct experiments using LLMs on multiple task datasets, including tasks such as JSON, Mermaid flowchart, and function call expression generation. These experiments show that the approach in this paper improves its accuracy by a factor of 1.09 to 11.6 compared to the benchmarks, and in the best case is able to reduce the number of samples used by the LLMs to generate JSON to about 16.5% of the benchmarks, which will effectively improve the usability of the content generated by the LLMs for computer programs.

Key Words: Large language models, Structured content generation, Computer-aided programming, Constrained decoding, Coroutine, Context-free grammar

目录

1 绪论

1.1 选题背景及意义

LLMs 已经让人们见证了计算机拥有一定程度上的理解和生成自然语言的能力，并且对不同领域的任务均有一定泛化能力，这种人机对话能力可以帮助人类解决各种问题。然而，由开发人员所编写的计算机程序却不像人类那样拥有极强的鲁棒性。在合理输入提示词的前提下，有两种路径让预训练 LLMs 更准确地指导完成一系列自动化任务，或作为计算机程序中间件纳入更多种类的生产过程。

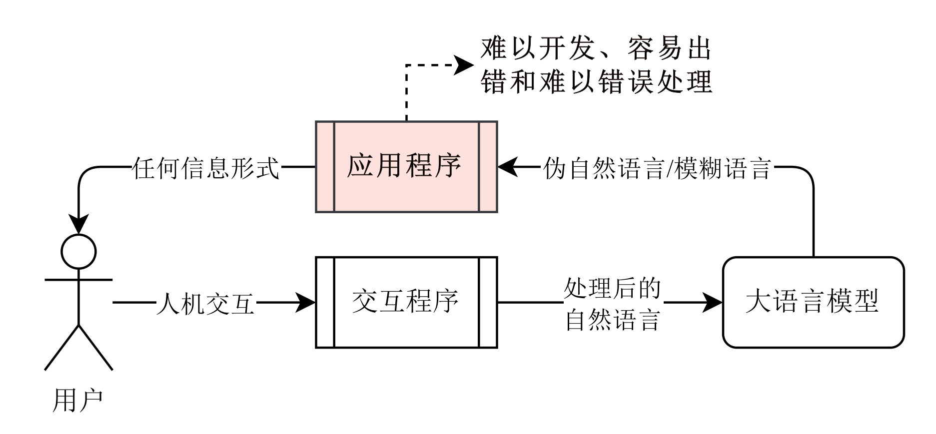


图 1-1 语言模型可能无法输出预期结构的信息以被程序解析

这两种常用途径分别是开发鲁棒性更好的应用程序和对齐语言模型。但是这两种途径可能都存在一定的问题，具体来说：

(1) GPT 等语言模型所拥有语言理解和生成能力使得人们能够开发出更接近通用人工智能的应用程序。但是如果需要基于大语言模型开发如图 1-1 所示的应用程序，则开发者经常会遇到一些痛点。大语言模型的输出是依赖于概率的，用户或计算机程序无法 100% 确保大语言模型生成的内容符合预期的结构。开发者开发鲁棒性更高程序去应对模型的输出更为困难、成本很高。模型的输出不符合预期往往带来更多信息丢失、计算资源浪费、甚至整个任务链路的失败。

(2) 另一种方案是对齐语言模型，使得此模型在特定任务上更适配我们的应用程序。这样也能够提升生成内容的可用性，并减少计算资源的浪费。然而，对齐语言模型会带来"对齐税"（大语言模型在预训练期间获得了广泛的能力，但在人类反馈的强化学习下对齐模型会导致其遗忘[1]），针对某一类任务新对齐的模型还需要新的部署，导致所需的计算资源成本大大提升。

1.2 国内外研究现状

语言模型不仅仅能够直接与人类进行对话，还应该拥有准确使用给定工具的能力，这在形式上等价于让语言模型理解指令，然后使用给定的应用程序接口。而这本质又要求语言模型输出可以被计算机解析为符合预期数据结构的领域特定语言。无论是学术界还是工业界，相关工程和研究近年来在不断出现。

1.2.1 学术界对控制语言模型的研究

OpenAI 在 2018 年 2 月以一篇名为《通过生成式预训练提高语言理解能力》的论文发布了开源的预训练语言模型 GPT-1[2]，随后在 2019 年、2020 年分别发布了 GPT-2 和 GPT-3。此前，这三个版本仅仅在学术界有知名度，直到 OpenAI 于 2022 年 11 月 30 日发布了商业产品 ChatGPT。在随后的短短 5 天时间内，其注册用户达到了 100 万，两个月后其月活用户数量就达到了一亿，成为商业界和科技圈的热门话题。OpenAI 开放了网络接口，允许开发者使用 OpenAI 的语言模型来开发“更不可思议”的应用程序。更多研究人员参与到大语言模型的应用研究当中，他们发现通过一种提示技术，能够使语言模型通过“思维链”进行“推理”，并能够通过使用预定义工具集中的工具（如能够搜索互联网）来 “行动”。实验证明，这种组合极大地提高了输出文本质量，并赋予大语言模型正确解决问题的能力[3]。

OpenAI 注意到了大量开发人员对 GPT 拥有更多更灵活控制的需求，于 2023 年 11 月 6 日的 OpenAI DevDay 正式发布 GPT-4 Turbo。OpenAI 的首席执行官 Sam Altman 在 Opening Keynote 中提到了 6 大更新，在第二节的“More Control”部分重点推出 JSON mode。这是一个让 GPT 直接生成 JSON (JavaScript Object Notation) 文本的功能。不像模糊的自然语言，JSON 属于上下文无关语言，被工业界广泛采用，易于被程序解析为计算机中确定的数据结构，从而更好地参与程序的逻辑链路。

ElementAI 的研究人员发现，采用增量解析来约束语言模型的自回归解码 (Parsing Incrementally for Constrained Auto-Regressive Decoding, PICARD) 能够有效的让语言模型生成形式语言，尤其是生成有效的 SQL 查询语句[4]。这种让语言模型在自回归解码过程中拒绝掉不合法的词，仅输出能够构成合法 DSL 前缀的约束解码，可能正是 OpenAI 实现 JSON mode 的思路[5]。除了 JSON mode 之外，方法调用 (Function calling) 也能从 GPT 中获取结构化输出。通过预先告知的方法名、方法描述和调用所需的参数和对应类型，GPT 会生成 JSON 格式的函数调用信息，其形式上仍然是一种 “JSON mode” ，但是这使得 GPT 拥有了使用工具的能力。

Google 和 MIT 的研究人员于 2023 年 5 月底提出了 Grammar Prompting，先让 LLM 根据上下文生成 BNF 的语法规则，然后再用约束解码让 LLMs 根据语法规则生成文本。通过语义解析 DSL (SMCalFlow, GeoQuery, Overnight)、动作 DSL (PDDL planning)、分子生成 (SMILES) 等实验，研究人员认为 Grammar Prompting 可以改进标准提示基线，这种方法还有望更好地帮助 LLMs 使用工具[6]。

1.2.2 工业界对生成结构化串的需求

在工业界，让大语言模型生成结构化文本的一个典型需求是生成 JSON 字符串，一些开发人员意识到 LLMs 并不能稳定地生成有效的 JSON 串，但幸运的是 LLMs 犯错的情况总是比较典型，因此可以通过一个解析程序在大部分情况下不破坏含义地修复 LLMs 产生的错误[7]。这种方法属于先让 LLMs 生成内容，然后再尝试修复它，程序并不一定能适配各种大模型并在所有情况下修复字符串。

来自微软的开发人员于 2022 年 11 月 11 日在代码托管平台 GitHub 提交了 guidance 的第一行代码，这是一个用于控制大语言模型的 Python 软件包[8]。通过不断地更新迭代，guidance 提供了使用正则表达式、上下文无关文法和交织控制 (Interleave Control) 等的方式来控制来自 Transformers, LlamaCpp, VertexAI 和 OpenAI 等接口的语言模型[8]，使开发人员能够更灵活的控制语言模型。

由 Huggingface 在 GitHub 开源的 Transformers 库，提供了大量的预训练语言模型和源代码实现[9]。来自 EPFL 的开发人员 Saibo Geng 于 2023 年 11 月 17 日在 Transformers 仓库提交的 PR (Pull Request) #27557 提供了一个使用 EBNF 接口的草案"上下文无关文法的约束解码"代码实现。Saibo Geng 认为语法约束的语言模型明显优于不受约束的语言模型，甚至击败了特定任务的微调模型。语法约束对于利用现成的语言模型进行广泛的结构化 NLP 任务具有很大的前景，尤其是在训练数据稀缺或微调成本高昂的情况下[10]。

工业界更常用的 vLLM 库提供了速度快且灵活的开源 LLMs 推理服务实现[11]。开发人员 Andrew Lapp[1] 于 2023 年 12 月 14 日在 vllm 仓库提交了 PR #2105 实现了一个增量的 LALR 算法的 CFG 或正则解析器，用以在语言模型生成过程中确定 "legal-next-token" 集合。此 PR 依赖于在 GitHub 开源的语法解析器 Lark[12]，实现了 EBNF 接口的 `vllm.grammar.GrammarLogitsProcessor` 后处理器，用以约束大语言模型生成形式上严格符合约定语法的内容。

[1]https://github.com/lapp0

1.2.3 现有研究和技术方案的局限性

现有研究和技术方案的问题主要可以描述为三点，分别是训练和推理成本很高、方法泛用性和易用性不好、方法鲁棒性和时间复杂度不好。

其一：LLMs 的训练和推理成本很高，针对不同的 DSL 生成需求训练或调优模型过程繁琐、成本很高，且仍无法期望生成内容达到 100% 的可用性。如果生成内容的语法不符合预期，不能通过程序的语法检查，就只能重新生成或以失败终止。工业界针对 JSON 的修复方法不能处理所有情况，且不适用于其他 DSL ，开发人员编写在这类"修复"程序的心智负担较大。

其二：Antlr4、Lark 和 Tree-sitter[13] 等 Parser Generator 的设计目标是生成一个高性能解析（或增量解析）符合文法的串为 AST 的程序源代码。他们的设计目标不是提升鲁棒性或用于自动化生成 DSL。当输入的文本不符合文法时，这些程序的解析过程通常会出错并终止，可以获取出错信息，但通常位置和原因不够精确，不足以引导生成 DSL。这些解析器也无法从解析中途恢复并指导 DSL 生成。

其三：Transformers 和 vLLM 仓库的 CFG 相关 PR 提案，以及上述的期刊、会议的相关研究，采用的方法并非"异步" + "指令"方法。即便是增量解析，在最差的情况下，现有技术在自回归语言模型生成每一个 token 的时候都需要验证一遍当前已生成的前缀是否属于 DSL 或 DSL 的前缀，一些优化方案是增加方法调用缓存以及增量解析，但时间复杂度仍然可能不好。生成长度为 n 的 DSL，若验证 DSL 的时间复杂度为 $O(f(n))$ ，那么生成 DSL 的复杂度将会是 $\sum_{i=1}^{n} O(f(i))$ 。根据 DSL 或解析器的不同，$O(f(n))$ 可能是 $O(n), O(n^2)$ 或 $O(n^3)$ 等情况。

1.3 课题研究内容

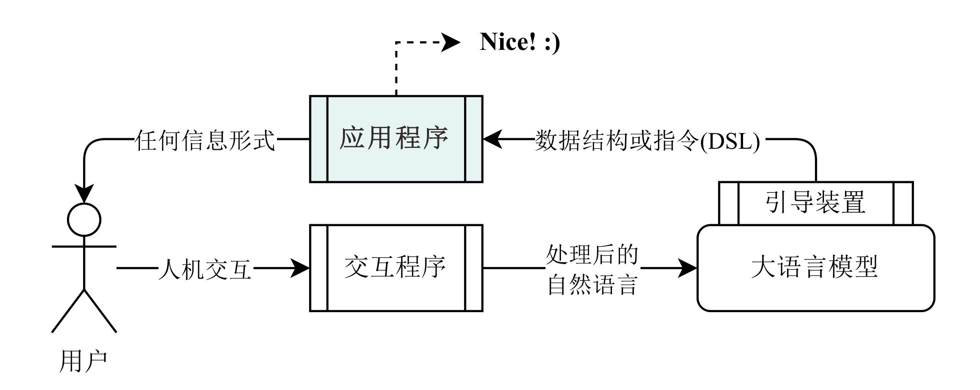


图 1-2 引导产生数据结构或指令使得应用程序更好工作

首先，开发人员需要以更低的心智负担，将语言模型纳入到应用程序实现中。在图 1-2 这个模型中，用户既不会直接与 LLMs 对话（可以减少 LLMs 被攻击者“洗脑”的不安全性），也不会直接接受 LLMs 的输出，而是经由应用程序的处理转换为其他人机交互的体现（例如通过图表、动画和声音等信息形式响应用户）。

其次，为应对现有研究和技术方案的局限性，本研究设计的引导装置包含了两个子装置：装置一是采用异步和协程的 DSL 解析和产生装置，即“异步 DSL 解析语言生成装置”；装置二是在装置一的基础上，调用语言模型进行 DSL 采样的解码装置，即“语言模型引导装置”。通过上述两个装置，实现引导 LLMs 在合适的位置产生预期的 DSL，开发人员得以更轻松地利用 LLMs 的能力开发鲁棒性更高的应用程序。整个系统大致流程如图 1-3 所示。

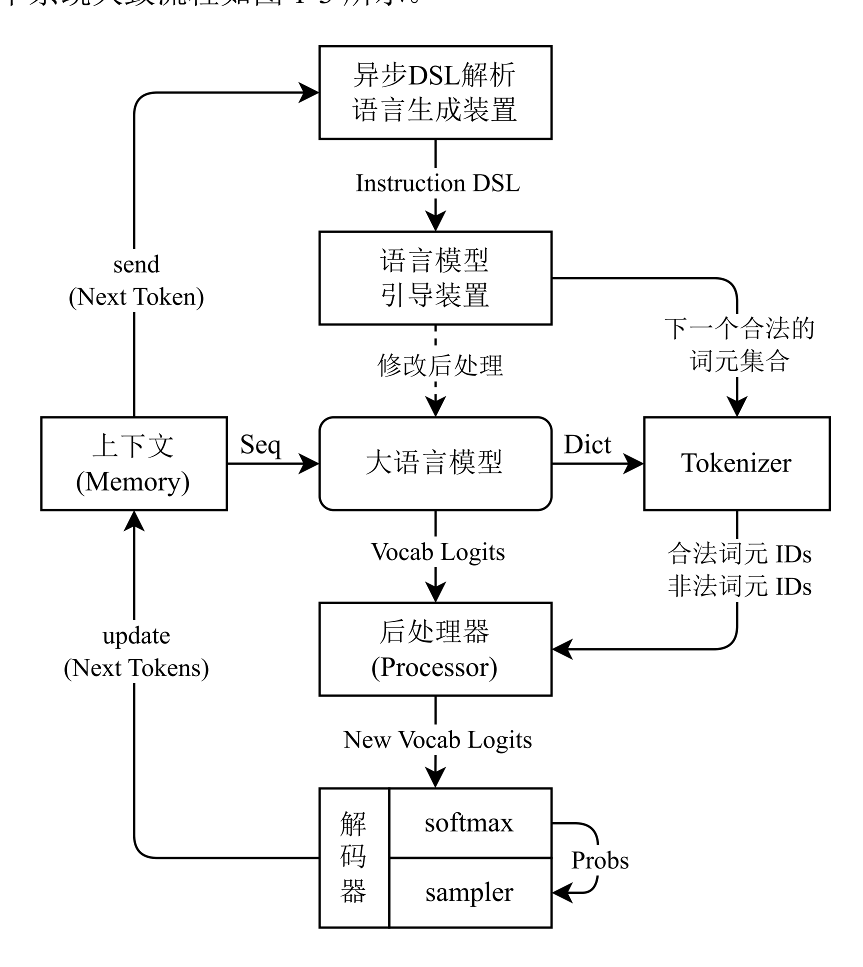


图 1-3　引导大语言模型生成可解析内容的系统

(1) 开发人员基于本研究实现的框架和其提供的若干组合子方法、装饰器和生成器，用 Python 协程、类、方法和递归等编程概念编写 BNF 的上下文无关文法语言的生成器类。此生成器能够产生一种用以产生给定 CFG 对应语言的引导指令。

(2) 装置一运行开发人员给定的生成器类，产生引导指令并驱动装置二，装置二驱动自回归语言模型从语言引导指令中采样下一个合法的词元。具体来说，对于 Transformer 架构的模型，装置二会在解码之前的“后处理”阶段将所有不合法词元（语言模型词汇表对应的词元 ID 类型）禁用，然后再由语言模型的解码装置产生下一个词元（字符串类型），同时选取能合法构成下一个 DSL 前缀的词元前缀。

(3) 产生的下一个词元前缀会被纳入自回归语言模型的上下文当中，同时此词元前缀也会输入到装置一，使得其产生下一个引导指令。直到装置一产生结束指令（例如 EOF、EOS 等指令），DSL 生成完毕。

1.4 论文结构安排

本文将主要介绍工业界常用的领域特定语言的结构和产生方式，如何设计一个采用协程实现的异步 DSL 产生和解析装置，以及如何在此装置基础上为自回归语言模型（尤其是 Transformer 架构的语言模型）构建一个约束解码装置，使得开发人员可以在此装置基础上更轻松地利用大语言模型的能力开发鲁棒性更高的应用程序。具体来说，各个章节内容安排如下：

第一章：绪论。主要内容是本研究的背景、大语言模型纳入应用程序存在的困难、学术界对控制语言模型的研究、工业界对生成结构化串的需求、现有研究和技术方案的局限性、课题研究内容和本论文的结构安排。

第二章：开发人员需要语言模型生成 DSL 开发应用。主要介绍了工业界、学术界和其他领域常用的 DSL 形式。计算机依赖 DSL 来实现具体功能，而开发人员既需要依赖 DSL 也需要利用语言模型的通用能力来开发应用。

第三章：大语言模型在 DSL 生成任务上的性能缺陷。主要介绍了现有大语言模型在按照指令要求生成 DSL 时性能较差，不能总是能够生成符合预期结构的文本。本章还从现有语言模型的解码策略上分析了产生这种性能缺陷的原因。

第四章：产生 DSL 的基本规则和范式，实现生成器。主要介绍了形式语言的产生规则，将 DSL 认为是一种形式语言，尤其是最常用的上下文无关语言。本章介绍了采用递归下降和协程实现的 DSL 生成装置，此装置也可用于 DSL 解析和验证。通过随机采样或其他采样方法，此装置能够产生计算机可以解析的内容。

第五章：生成开发者预期的 DSL 通过语言模型采样。主要介绍了基于约束解码的方法将现有 Transformer 架构的语言模型用作采样器，在生成器中选择 DSL 产生路径。此方法也可以用于其他自回归语言模型。

第六章：本研究在语言模型生成DSL领域取得优势。主要介绍了本研究的实验结果，在多个任务数据集上进行实验，包括JSON、Mermaid流图和函数调用表达式等DSL生成任务，以及测试了DSL约束装置对语言模型采样次数的影响。

第七章：总结与展望。对本研究所做的工作进行总结，展望了本研究所提出的方法的用途，并说明本研究的局限性及可能的研究方向。

1.5 本章小结

本章主要介绍了开发人员在使用大语言模型开发具体应用程序时可能面临的困难：难以开发足够高鲁棒性的计算机程序将语言模型的输出内容解析为开发人员和程序所期望的数据结构。学术界和工业界现有三种途径来解决这一问题，包括“事后修复”、“对齐模型”和“约束解码”。然而，现有方法均存在一定的局限性，本研究将改进现有的“约束解码”实现，分别实现异步DSL解析语言生成装置和语言模型引导装置，开发人员能在此基础上更容易的利用语言模型产生DSL以开发鲁棒性更高的应用程序。

2　开发人员需要语言模型生成 DSL 开发应用

领域特定语言已经在编程实践中得到了非常广泛的应用，开发人员依赖 DSL 进行数据交换、配置文件编写和业务逻辑实现。DSL 的解析和产生是各种编程语言的标准库、软件包或应用程序开发框架广泛提供的功能。随着大语言模型的发展，人们发现大语言模型在语义理解和生成方面拥有广泛的用途，开发人员希望借助 LLMs 的通用能力开发新的应用程序，而 DSL 正是 LLMs 与现有应用程序交互的一种重要途径。因此，开发人员需要语言模型生成 DSL 开发应用。

2.1　领域特定语言

领域特定语言 (Domain-Specific Language, DSL) 是一种为特定领域应用程序设计的计算机语言，也被称作面向应用语言[14]、特殊用途语言[15] 等。与通用编程语言 (General-Purpose Language, GPL) 不同，DSL 专注于特定领域的表达能力，并提供更简洁、更易于理解的语法和语义。

为方便描述，本文所指的 DSL 强调可被计算机程序解析为特定数据结构的串，或称为形式语言 (Formal language)，不区分 DSL 和 GPL。用于数据交换的 JSON、XML、YAML 等语言，用于查询的 SQL、XPath、XQuery 等语言，用于排版的 HTML、$\LaTeX$、Markdown 等语言以及各种编程语言都认为是一种 DSL。

2.2　工业界常用的 DSL

一些软件开发人员除了需要使用通用编程语言（例如 C、Java、Go 和 Rust 等）外，在开发过程中还需要使用各种各样的 DSL。这些 DSL 可能涉及包括标记语言、信息交换语言、数据格式、应用程序接口、软件配置、图文排版等各种领域。表 2-1 列出了部分工业界常用的 DSL 名称、MIME 类型和其简介。

表 2-1　部分工业界 DSL 举例

名称	MIME 类型	简要介绍
JSON	`application/json`[16]	JSON 定义了一套轻量规则，用于可移植地表示结构化数据。JSON 的文法是上下文无关的，是 Web 最常用的数据交换格式之一，被广泛应用于前后端数据交换、配置文件编写（例如 NodeJS 的 `package.json` 文件）等场景。

名称	MIME 类型	简要介绍
XML	`application/xml`[17]	可扩展标记语言 (eXtensible Markup Language) 是一种标记语言，可用于存储、传输和重构松散数据。XML 的文法是上下文无关的，被广泛应用于前后端数据交换、配置文件编写（例如 Maven 软件包的 `pom.xml` 文件）等场景。
HTML	`text/html`[18]	超文本标记语言 (HyperText Markup Language) 是一种用于创建网页的标记语言。HTML 的文法是上下文无关的，广泛应用于排版和前端应用开发。
CSV	`text/csv`[19]	逗号分隔值 (Comma-Separated Values) 是一种通用的、简单的文件格式，以纯文本存储数字或文本的表格数据。其作为信息交换格式被用户、商业和科学界广泛应用。
Mermaid[1]	`text/vnd.mermaid`[2]	基于 JavaScript 的图表创建工具，它的语法和渲染受 Markdown 启发，用于创建和修改各种复杂的图表，包括流程图、时序图、甘特图、类图、Git 图、实体关系图等[20]。

表 2-1 中的 MIME 指的是互联网媒体类型 (Multipurpose Internet Mail Extensions, MIME)，是一种互联网传输内容类型的标准，也是一种被 BNF 定义的 DSL，最初为互联网请求意见稿 (Request for Comments, RFCs) 所定义[21]。

2.3 学术界使用的 DSL

学术界最常见的需求是是需要一种 DSL 来表示数学或其他学科领域的公式，这种 DSL 应该是准确、严格且易于用户输入的。它能够被相应的计算机程序解析为特定数据结构，然后再由计算机排版和渲染算法将其转换为矢量图或位图，以便用于屏幕显示或印刷。能够处理复杂数学符号排版的最经典 DSL 便是 TEX 文本，其次还有被 HTML5 标准兼容的 MathML（一种用于描述数学符号和捕获它的结构与内容的标记语言[22]）和新兴的用于排版的 Typst 编程语言[23]。

在物理领域，Q# 是由微软开发的一个量子算法工具包，后来芝加哥大学的 Kartik Singhal 等人给予了它正式的规范，提出的 λ-Q# 拥有更严格的数学定义和类型系统[24]，这使得在描述量子算法上也拥有了良好句法定义的 DSL。

一些研究人员创造了专用的标记方法 SMILES 来表示化学分子式[25]。这种表示方法使用 ASCII 字符串明确分子结构，一些研究人员使用深度神经网络训练生成

[1]https://mermaid.js.org/intro/

[2]https://www.iana.org/assignments/media-types/application/vnd.mermaid

SMILES 字符串的概率模型来产生化学结构[26]。2007 年开源化学社区开发了开放规范 OpenSMILES[1]，社区使用了 BNF 形式的文法定义了这种语言[27]。

在生物领域，科研人员使用深度学习、自然语言处理等技术来研究蛋白质，他们认为常见蛋白质的基序和结构域类似于人类语言中的单词、短语和句子[28]，因此 DSL 也可用作研究蛋白质的工具。

2.4 其他领域需要 DSL

除了工业界和学术界会使用 DSL 外，在其他一些领域（例如商业、艺术等领域）也会用到 DSL。专用于商业的编程语言 COBOL (Common Business Oriented Language) 是最早的高级编程语言，也是世界上最早实施标准化的计算机语言之一。专用于实时声音合成或音乐创作的计算机编程语言 ChucK[29]、基于 XML 的声音计算系统 Csound[30] 和为电子艺术以及视觉交互设计而创建的编程语言 Processing[31] 等诸多 DSL 也在其各自领域得到了广泛应用。

2.5 语言模型生成 DSL

GPT 等语言模型已经让开发人员见证了其在各个领域的通用能力，包括机器翻译、文本摘要、信息提取、问答助手、代码生成等等能力。更具体的，我们还希望语言模型能够在具体的环境下做识别、分类、路由、推理、决策等任务。当语言模型参与到具体的计算机应用或研究当中时，需要生成特定的 DSL 以便计算机程序识别。如果大语言模型能够遵循指令要求生成 DSL，那么其易用性和泛用性将会大大提高。这将是让人工智能学会使用工具并走向通用人工智能的途径之一。

2.6 本章小结

本章介绍了在工业界、学术界、商业界、艺术界等与计算机交汇的领域，DSL 得到了极为广泛的应用，并且已经有不少研究人员采用语言模型来进行学术研究。GPT 等语言模型拥有强大的泛用性，而开发人员依赖 DSL，如果语言模型能够拥有准确遵循指令而生成开发人员所期望的 DSL 的能力，那么语言模型将更加实用且拥有极为广泛的应用前景。

---

[1]http://opensmiles.org/opensmiles.html

3　大语言模型在 DSL 生成任务上的性能缺陷

虽然 DSL 在各个领域应用广泛，但是基于深度学习的模型产生合法的 DSL 是一个有挑战性的任务。例如 DeepSMILES 的作者在 2018 年的论文中提到，如果使用一个基于字符级别的循环神经网络 (RNN) 模型来生成 SMILES 字符串，这个模型可能生成大量不合法的字符串，原因在于模型总是不能产生匹配的括号对[26,32]。这种不合法的字符串无法被化学软件解析为合法的化学结构。

众多 DSL 在文法层面规定了匹配的括号对、引号对或其他语言结构，对应的软件也依赖于这些结构来解析 DSL。如果需要用 LLMs 生成 DSL，就需要先验证现有的 LLMs 是否能够根据指令生成约定的 DSL，并且其性能如何。

3.1　常见的 DSL 句法规则

最常见的句法就是匹配的括号对，其基本规则是每一个左括号的右边都有一个右括号与之匹配。这样的语法结构在众多编程语言或 DSL 中都有应用，例如 JavaScript、Python、Lisp 和 JSON 等。如果字母表 Σ 仅包含 '(' 和 ')' 两个字符，那么最简单的括号对是 '()'。给定一个字符串 $s \in \Sigma^*$，一种常用来验证此字符串是否属于合法括号对的算法是使用栈来验证，但对于仅有两种字符的情况，可以简化为公式 (3-1) 所示的函数 $\mathrm{Legal}(s)$ 来验证 s 是合法的字符串前缀。

$$\mathrm{Legal}(s) = \bigwedge_{k=1}^{|s|} \left[\left(\sum_{i=1}^{k} \begin{cases} +1 \text{ if } s[i] = \text{'('} \\ -1 \text{ if } s[i] = \text{')'} \end{cases} \right) \geq 0 \right] \tag{3-1}$$

当字母表 Σ 包含多种括号对，例如 $\Sigma = \mathbb{L} \cup \mathbb{R}$; $\mathbb{L} = \{$'('，'['，'{'$\}$; $\mathbb{R} = \{$')'，']'，'}'$\}$，如果所有合法的 $s \in \Sigma^*$ 构成集合 $L(G)$，那么图 3-1 所示的算法可以验证一个字符串 s 是否属于语言 $L(G)$，其中 G 是语法，ε 是空串。

```
LEGAL-BRACKET-PAIRS (s ∈ Σ*, m: 𝕃 ↦ ℝ):
σ ← []                              ▷ 初始化空栈 σ
for c in s:                         ▷ 按顺序遍历 ∀(i, c) ∈ s
    if c in m:                      ▷ 字符 c 是左括号
        push c to σ                 ▷ 将字符入栈 σ
    else:                           ▷ 字符 c 是其他字符
        l ← (pop σ) if |σ| > 0 else ε
        if m(l) ≠ c :               ▷ c 不能与 l ∈ 𝕃 匹配
            return False            ▷ s 不是合法字符串
return |σ| ≯ 0                      ▷ ∀(i, l ∈ 𝕃) ∈ s 被匹配则合法
```

图 3-1　检查括号对是否匹配的算法

除了括号对外，字符串转义也是各种DSL中常见的句法规则。当用一对引号包裹一串字符时，这一串字符就不能直接的包含引号，而是需要转义。如果表示包含一个 " 字符的字符串，则需要写成 "\"" 而不是 """ ，否则应用程序的语法解析器会在第二个 " 字符处理完毕后结束字符串的解析而导致信息损失。

3.2 验证 DSL 生成性能的实验思路

既然匹配的括号对和字符串转义是 DSL 中广泛存在的句法规则，那么要验证语言模型的DSL生成性能，就可以先从这两方面入手。如果语言模型连这些最基本的句法规则都无法遵循，那么生成更为具体的 DSL 的性能显然不会很好。

由于GPT等常见语言模型是自回归模型，其生成的文本是逐个词元生成的，那么对于这些语言模型，他们要生成合法括号对 $L(G)$ 的前提是生成合法括号对字符串的前缀。假设语言模型仅预测的词元是 $l=$ '(' 和 $r=$ ')' ，根据公式 (3-1) 和合法括号对的规则可知 $\mathbb{I}=\{srx\ |\forall s\in L(G),x\in\Sigma^*\}$ 是非法的括号对集合。

$((((()))))\ x \rightarrow$

(	97.8%
)	2.2%

$(((((())))))\ x \rightarrow$

(	99.9%
)	0.1%

$((((((())))))))\ x \rightarrow$

(	97.4%
)	0.6%

$(((((((()))))))))\ x \rightarrow$

(	97.9%
)	2.1%

图 3-2 让语言模型 (GPT-2) 预测下一个括号的概率

实验的基本思路是构建字符串 $s=l^n r^n\in L(G)$ ，输入表 3-1 的提示词让模型预测 '(' 或 ')' 的概率，而 $r=$')' 的概率恰好就是错误率，就如图 3-2 所示意。

表 3-1 让 LLMs 补全括号对字符串的提示词

中文提示词	英文提示词
给定一个仅包含 (,) 的字符串。	Given a string containing only (,).
有效的括号对字符串必须满足：	A valid bracket pair string must satisfy:
1. 每个左括号的右侧都有一个与之匹配的唯一对应的右括号。	1. The right side of each left bracket has a unique corresponding right bracket that matches it.
2. 不同位置的左括号与不同位置的右括号相匹配。	2. Left brackets in different positions match right brackets in different positions.
有效的括号对字符串：	A valid bracket pair string:
`` + \n + "(" * n + ")" * n [1]	``` + \n + "(" * n + ")" * n

[1]这里的意思是提示词的末尾连接换行符和字符串 $l^n r^n$

3.3 验证语言模型的 DSL 生成性能

通过给语言模型提示词和 $l^n r^n \in L(G)$，让其生成下一个词元 x，计算 $x = l$ 或 $x = r$ 的概率，就可以得到语言模型在合法括号对生成任务上的性能。图 3-3 展示了在四种模型上实验，随着 $|s|$ 的增加，生成 r 和不合法串的概率的变化。

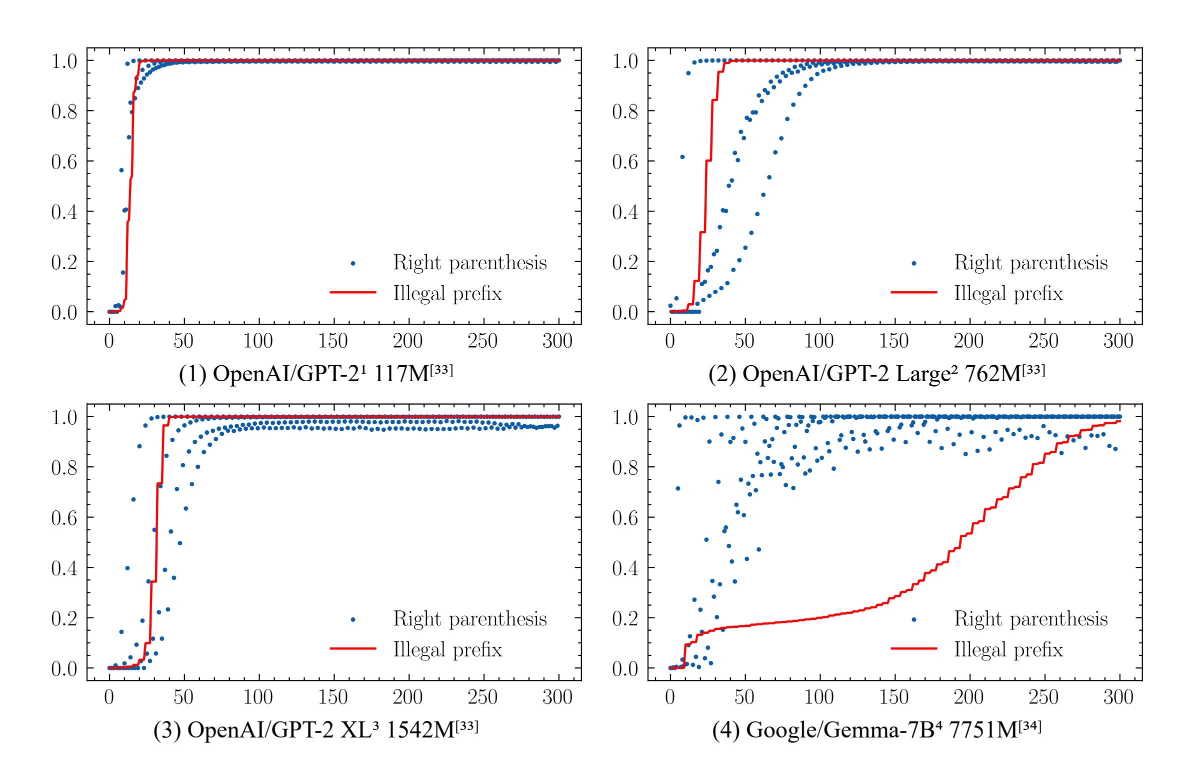


图 3-3 模型生成错误 DSL 的概率随生成长度而增大

图 3-3 的数据显示了四种模型在 $|s|$ 增加时，错误率 $r(e)$ 不断提升，也就是说在括号对这种基本 DSL 生成任务下，其 DSL 生成性能也在快速下降。

表 3-2 错误率 $r(e)$ 与平均字符串长度 $|s|$ 的关系

模型名称	参数量	Date^{GMT+8}	$\|s\|_{r(e)>50\%}$	$\|s\|_{r(e)>95\%}$
GPT-2	117M	2019-2-14	14	20
GPT-2 Large	762M	2019-2-14	24	32
GPT-2 XL	1542M	2019-2-14	32	36
Gemma	7751M	2024-2-21	194	282

表 3-2 表明，参数量达到一亿的 GPT-2 在生成字符串长度达到 20 时错误率就飙升到 95%，即使是参数量达到近 78 亿的 Gemma 模型，其 $|s|$ 达到 282 时，错误率也高达 95% 以上。这些语言模型的 DSL 生成性能是不可接受的。

[1] https://huggingface.co/openai-community/gpt2
[2] https://huggingface.co/openai-community/gpt2-large
[3] https://huggingface.co/openai-community/gpt2-xl
[4] https://huggingface.co/google/gemma-7b

3.4 让语言模型生成 DSL 是困难的

在一定任务长度下，神经网络可以正常工作，但随着长度增加，神经网络失效是一种非常典型的事情。人类“一目了然”的情况，神经网络可以解决。但是，如果需要做一些“更算法化”的事情（例如明确地计算括号以查看它们是否匹配），神经网络往往在某种程度上因为“计算太浅”而无法可靠地完成，就算是当前完整 1750 亿参数的 ChatGPT 也很难正确匹配长序列中的括号对[32]。

大语言模型通常使用自回归模型来生成文本。在自回归模型中，下一个词的概率分布取决于前面的词。然而，如果每次都选择概率最高的词，模型将过于“保守”而缺乏创意和多样性。在大语言模型工业实践当中，不同的采样器可以用来帮助语言模型提升性能。常见的采样方法有随机采样、贪婪搜索 (Greedy search)、束搜索 (Beam search) 和核采样 (Nucleus sampling) 等方法。

LLMs 采样之前还可以进行例如“温度”的后处理操作来改变模型输出层 Logits 张量中的概率分布，以影响采样。不同的采样参数是语言模型的超参数，这些超参数可能会影响 LLMs 的输出和性能表现[35,37]。大语言模型的温度参数是在工程实践中经常调节的参数之一，其值是大于或等于 0 的浮点数，通常介于 $[0,1]$ 之间（大于 1 会导致模型性能快速下降[38]），用于调整模型输出的多样性。

温度采样与玻尔兹曼分布有关，它给出一个系统处于某种状态的概率，如公式 (3-2) 中的 ρ_i 描述了该状态的能量及温度的概率测度函数。

$$\rho_i = \frac{1}{Q} e^{-\varepsilon_i/(kT)} = \frac{e^{-\varepsilon_i/(kT)}}{\sum_{j=1}^{M} e^{-\varepsilon_j/(kT)}} \tag{3-2}$$

公式 (3-2) 与公式 (3-3) 的 softmax 函数非常相似，只不过 softmax 函数没有在向量 z_i 上除以 $-kT$ 。温度 T 越高，ρ_i 的分布越均匀，而 T 越低，ρ_i 的分布越集中，就如图 3-4 所示。温度采样通过调整参数 T 来改变词元的概率分布，增大 T 可以显著增加大语言模型输出内容的多样性。

$$\mathrm{softmax}\left(-\frac{z_i}{T}\right) = \frac{e^{-z_i/T}}{\sum_{j=1}^{M} e^{-z_j/T}} \tag{3-3}$$

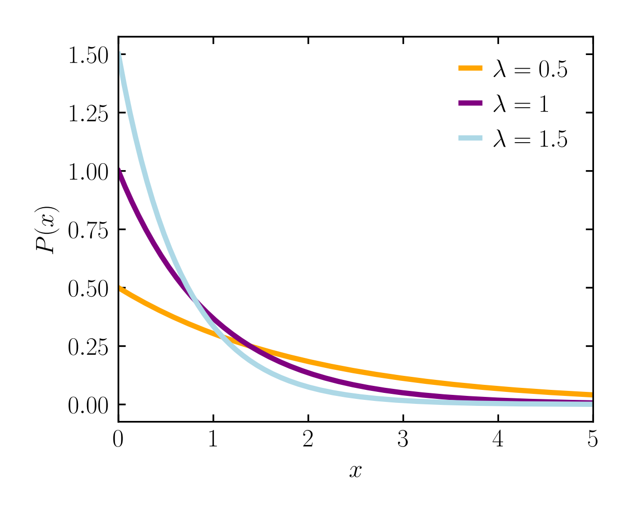


图 3-4　指数分布的概率密度函数图[39]

玻尔兹曼分布是一种 $P(x) \sim \mathrm{Exp}(x)$ 的指数分布。如图 3-4 所示，当 λ 从 1.5 降低到 0.5 时，分布越来越平缓，这意味着温度提高会使得其他可能性的词更有可能被选中。特别地，温度为 0 时相当于选择概率最大的 Top-1 词元。

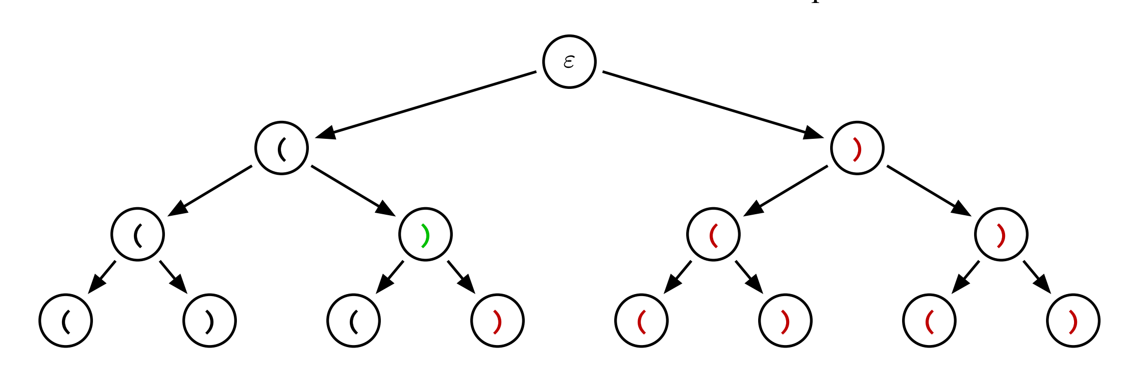


图 3-5　合法括号对字符串的树状示意图

自回归语言模型生成一串文本，相当于在一个巨大的单词表上进行采样。对于匹配括号对这样的任务来说，单词表可以仅有左右括号和 EOS 三个词元。语言模型从空串 ε 开始在如图 3-5 的树状图上“游走”，在正确的位置上“停下来”（生成 EOS 词元）是艰难的。因为正确位置非常稀少，错误路径数量总是超过正确的路径数量。由于 LLMs 的采样通常具有“温度”等随机性，只要走错一步就“前功尽弃”了。

3.5　本章小结

本章研究了 DSL 的典型句法特征“匹配括号对”，然后给出了匹配括号对的验证算法和规律。之后，本章介绍了验证语言模型生成 DSL 性能的实验思路。本章重点展示四种语言模型在括号对生成任务上的性能，结果显示这些语言模型在生成 DSL 任务上的性能是难以接受的。最后，本章从理论上简单解释了自回归语言模型在生成 DSL 任务上性能不佳的原因。

4　产生 DSL 的基本规则与范式、异步与协程

4.1　形式语言层次与基本概念

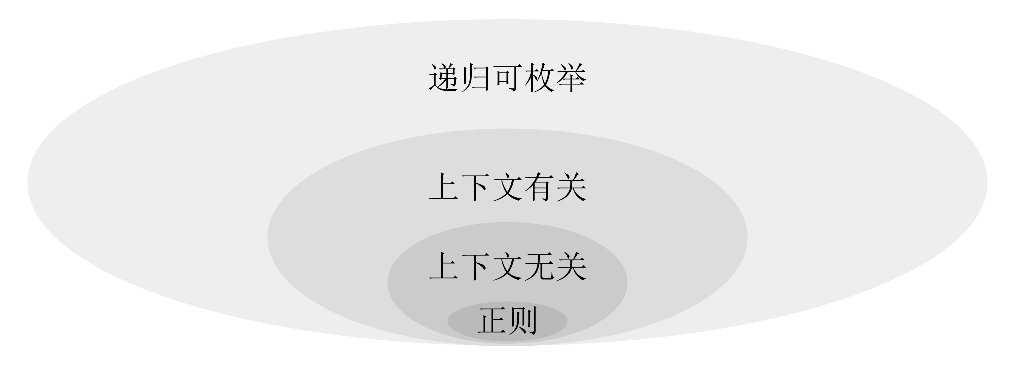


图 4-1　乔姆斯基谱系

语言学家诺姆·乔姆斯基 (Avram Noam Chomsky) 于 1956 年提出了著名的乔姆斯基谱系。乔姆斯基谱系是计算机科学中描述形式文法表达能力的一个分类谱系。它将形式语言分为四个层次[40]，并且逐级包含，如图 4-1 和表 4-1 所示。

表 4-1　乔姆斯基谱系的四种语法类型

语法类型	语言类型	可识别自动机	产生规则
0-型	递归可枚举	图灵机	$(\gamma \neq \varepsilon) \rightarrow \alpha$
1-型	上下文有关	线性有界自动机	$\alpha A\beta \rightarrow \alpha\gamma\beta$
2-型	上下文无关	下推自动机	$A \rightarrow \alpha$
3-型	正则	有限状态自动机	$A \rightarrow aB$

专门研究和表达语言语法的理论称作形式语言理论，而能被数学或计算机准确处理的语言被称作形式语言。领域特定语言作为一种形式语言，其语言也包含了两部分：语法和语义。首先，语法是 DSL 的基础，如果一个串的语法有问题，这个串就不能被约定的计算机程序所识别，并且也难以简单的修复或恢复信息。

形式语言定义在一个特定的字母表 Σ 上，例如集合 $\{\text{'('}, \text{')'}, \&\}$ 就是一个字母表（其中 & 表示结束符），集合 $\{1, 0\}$ 也可以是一个字母表。形式语言的字母表恰巧与大语言模型的词汇表相对应。

如果有一个字母表 Σ，字母表上长度为 n 的串记作 Σ^n，而任意长度的串记作 Σ^*，特别定义 Σ^0 是仅包含空串的集合 $\{\varepsilon\}$。字母表 Σ 上的语言 L 具有语法 G 表示为 $L(G) \subseteq \Sigma^*$。若 $a, b \in \Sigma^*$，ab 表示两者左右连接，a^k 表示 a 重复 k 次。

表 4-1 中乔姆斯基谱系中的上下文无关文法 (CFG) 拥有足够强的表达能力来表达绝大多数的编程语言，是描述编程语言语法的首选形式[41]。

4.2 产生式与上下文无关语言

形式语言的的语法由产生规则定义，产生规则包含了有限的非终结符 N 集 、有限的终结符集（字母表）Σ、有限的产生规则集 P 形如 $(\Sigma \cup N)^* N (\Sigma \cup N)^* \to (\Sigma \cup N)^*$ 和开始符号 $S \in N$ ，四元组 $G = (N, \Sigma, P, S)$ 定义了一个形式文法。

若 $l =$ '(' 且 $r =$ ')' ，那么可以公式 (4-1) 来表示合法的括号对序列。

$$P = \{S \to \varepsilon, S \to lSrS\} \tag{4-1}$$

公式 (4-1) 与公式 (4-2) 表达的含义相同，但是公式 (4-2) 可能更加清晰。

$$\begin{aligned} \langle \text{Pairs} \rangle &\to \langle \text{Pair} \rangle \\ \langle \text{Pairs} \rangle &\to \langle \text{Pair} \rangle \langle \text{Pairs} \rangle \\ \langle \text{Pair} \rangle &\to (\ \langle \text{Pairs} \rangle\ ) \\ \langle \text{Pair} \rangle &\to \varepsilon \end{aligned} \tag{4-2}$$

上下文无关语言作为编程语言语法的首选形式，当然也可以用来定义 C 语言。图 4-2 左侧展示了由 CFG 的产生规则定义的 C 语言的一个子集的语法。一串简单的 C 语言代码字符串 `if ( x > 9 ) { x = 0; y = y + 1; }` 可以被语法规则分解为如图 4-2 右侧的树状语法结构。

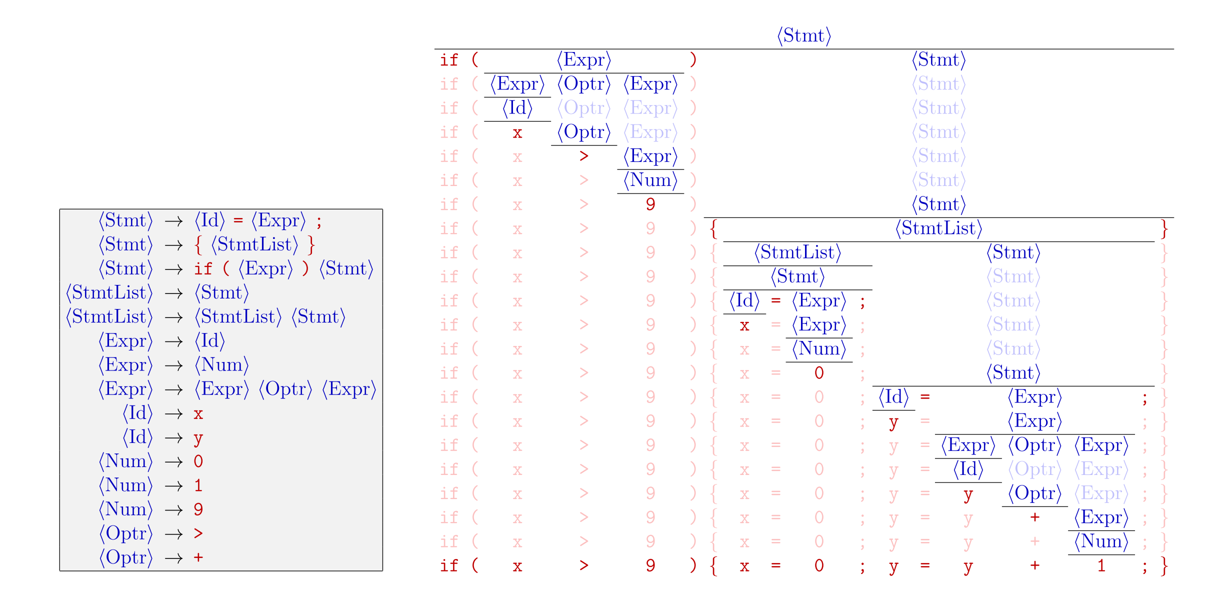


图 4-2 C 语言[42] 形式语法的一个子集和树状结构[43]

图 4-2 中的 ⟨Stmt⟩ 即表示一个语句，⟨StmtList⟩ 表示多个语句，⟨Expr⟩ 表示一个表达式，⟨Id⟩ 表示一个标识符，⟨Num⟩ 表示一个数字，⟨Optr⟩ 表示一个操作符。这些非终结符通过规则展开，可以构成一个合法的 C 语言代码字符串。

4.3 常见领域特定语言的规则

用于表达形式语言的一系列数学符号和延拓可以称为元语言，指的是讨论或研究语言本身时所用的语言。除了上述数学符号可以用来表达 DSL 外，巴科斯范式 (Backus Normal Form, BNF) 或其扩展 EBNF 也可以用于表达 DSL，并且其本质上与 Section 4.2 所述的形式语言语法 $G = (N, \Sigma, P, S)$ 等价。BNF 被广泛地用于编程语言、指令集和通信协议等领域，当然也包含本文 Section 2 所述的各种 DSL。

由于各种互联网技术规范都需要定义一个正式的语法规则，这种需求导致 BNF 的另一种增强形式 ABNF (Augmented BNF) 出现并已受到许多人的欢迎[44]。ABNF 是 BNF 的一个扩展，它平衡了紧凑性、简单性与合理性，适合用于描述 DSL 的语法，并成为一个互联网标准 (STD 68, RFC5234)[1] 。

```
{
  "title": "引导大语言模型生成计算机可解析内容",
  "author": "王家晔",
  "id": 20202005312,
  "keywords": [
    "Large language models",
    "Structured content generation",
    "Computer-aided programming",
    "Constrained decoding",
    "Coroutine",
    "Context-free grammar"
  ],
  "date": "2024-04-07"
}
```

图 4-3 一个 JSON 字符串样例

一个典型的 DSL 是 JSON 语言，一个样例如图 4-3 所示，它因为包含括号对且括号对能够“递归”的互相嵌套而不属于正则语言，是一种典型的上下文无关语言。JSON 语言的部分语法规则可以用如图 4-4 所示的元语言来描述。

```
JSON   := [ ws ] value [ ws ]
value  := false | null | true | object | array | number | string
object := '{' [ member *( ',' member ) ] '}'
array  := '[' [ JSON   *( ','   JSON ) ] ']'
number := [ '-' ] int [ frac ] [ exp ]
string := '"' *char '"'
member := [ ws ] string [ws] ':' JSON
exp    := ('e' | 'E') [ '-' | '+' ] 1*digit
int    := '0' | ( onenine *digit )
frac   := '.' 1*digit
```

图 4-4 JSON 的简要语法表示

[1] https://www.rfc-editor.org/info/std68

图 4-4 中的 ws 表示任意空白字符（例如 U+0020，U+000A，U+000D，U+0009 等），char 表示任意字符（如果歧义则需要转义），digit 表示数字 (0 到 9) 字符，onenine 表示非零 (1 到 9) 数字字符。除此之外，形如 [A] 的串表示 A 是可选的（即 A 或空串 ε ）；*A 表示 A 可以出现任意次；1*A 表示 A 至少出现一次；A | B 表示 A 或 B 之一；用单引号括起来的串 'A' 表示属于 Σ 的具体的终结符。

```
request-header := Accept
                | Accept-Charset
                | Accept-Encoding
                | Accept-Language
                | Authorization
                | Expect
                | From
                | Host
                | If-Match
                | If-Modified-Since
                | If-None-Match
                | If-Range
                | If-Unmodified-Since
                | Max-Forwards
                | Proxy-Authorization
                | Range
                | Referer
                | TE
                | User-Agent
```

图 4-5　HTTP/1.1 的请求标头字段的语法示意

除了 JSON 之外，超文本传输协议 (Hypertext Transfer Protocol, HTTP) 也可视为一种 DSL，因为 RFCs 用 ABNF 定义了它的文法[45]。HTTP 是为分布式、协作和超媒体信息系统而设计的应用程序级别协议，被全球互联网使用。

图 4-5 中的 Accept 、 Accept-Charset 和 Accept-Encoding 等都是 HTTP/1.1 请求标头字段的一部分，HTTP 的部分语法规则如图 4-5 所示，作为简单举例。

对于图文排版，Typst 就是一个为排版设计的可编程的标记语言和软件[23] 。图 4-6 展示了这个软件能够将 Typst 的 DSL 渲染为富文本并呈现给用户。同样，Typst 的 DSL 也能够由 BNF 来表达，图 4-7 呈现了 Typst 的部分语法规则作为参考。

```
#set par(leading: 0.5em)
- *Sine Function:*
  - $f(x)=sin(x)$\
- *Cosine Function:*
  - $f(x)=cos(x)$
```

(1) Typst DSL 样例

- **Sine Function:**
  - $f(x) = \sin(x)$
- **Cosine Function:**
  - $f(x) = \cos(x)$

(2) 处理后的富文本

图 4-6　Typst 将 DSL 渲染为富文本

```
linebreak   := '\' '+'?
text        := (!special)+
escape      := '\' special
quote       := "'" | '"'
strong      := '*' markup '*'
emph        := '_' markup '_'
raw         := '`' (raw | .*) '`'
link        := 'http' 's'? '://' (!space)*
math        := ('$' .* '$') | ('$[' .* ']$')
heading     := '='+ space markup
list        := '-' space markup
enum        := digit* '.' space markup
desc        := '/' space markup ':' space markup
label       := '<' ident '>'
ref         := '@' ident
markup-expr := block | ('#' hash-expr)
```

图 4-7 Typst 的基本语法示意[23]

4.4 递归下降的解析和生成器

为了让计算机程序能够识别 DSL，首先需要将 $s \in L(G)$ 中的各个子串分析为符号，然后根据产生规则 P 生成语法树。递归下降是一种常见的解析方法，它是一种自顶向下的解析方法，即从根节点开始，递归地向下解析，直到叶子节点。递归下降解析器的每个非终结符 $n \in N$ 对应一个可递归的函数。

包含左递归的形式语法不能被简单的递归下降解析器解析，除非它们被转换为弱等效的右递归形式。一些研究表明，通过使用“缩减”，可以在更复杂的自上而下的解析器中容纳左递归语法（以及所有其他形式的通用 CFG）[46]。

```
value = async function(set: Setter<JSONValue>) {
    const token = await this.token.read();
    switch (token.type) {
        case TokenType.LeftBrace:
            await this.token.unread(token);
            await this.object(set);
            break;
        case TokenType.LeftBracket:
            await this.token.unread(token);
            await this.array(set);
            break;
        case TokenType.Digits:
        case TokenType.Negtive:
        case TokenType.Positive:
            await this.token.unread(token);
            await this.number(set);
            break;
        case TokenType.Bool:
        case TokenType.Null:
        case TokenType.String:
            await set(token.value);
            break;
        default:
            console.error(`unexpected token: ${token}`);
    }
}
```

图 4-8 采用异步递归下降的解析 JSON 值的伪代码

如图 4-4 所示的 JSON 语法对应的语言作为一种上下文无关语言，当然可以被递归下降的解析方法解析为应用程序中的数据结构。图 4-8 展示了一个异步递归下降解析器的示例（基于 TypeScript 语言，但请当作伪代码参考）。

```
number = async function(set: Setter<number>) {
    // sign part (optional)
    const sign = await this.sign();
    // integer part
    const intToken = await this.matchToken(TokenType.Digits);
    let num = parseInt(intToken.value as string);
    // fraction part (optional)
    const dotToken = await this.matchOptionalToken(TokenType.Dot);
    if (dotToken !== undefined) {
        const intToken = await this.matchToken(TokenType.Digits);
        num += parseFloat(`0.${intToken.value}`);
    }
    // exponent part (optional)
    const eToken = await this.matchOptionalToken(TokenType.E);
    if (eToken !== undefined) {
        // sign part (optional)
        const sign = await this.sign();
        // exponent part
        const intToken = await this.matchToken(TokenType.Digits);
        const int = parseInt(intToken.value as string);
        num *= Math.pow(10, sign * int);
    }
    await set(sign * num);
}
```

图 4-9　采用异步递归下降的解析 JSON 数字的伪代码

采用递归下降的解析方法，通常需要一些工具函数，例如图 4-8 中所示的读词元 Token 的函数 `token.read()` 和回退词元的函数 `token.unread()`。这相当于有一个指针从头到尾的扫描待解析的字符串，在必要的时候还可以进行回溯。

如果将自顶向下的解析器反过来，从开始符号 S 开始，从语法 G 的产生规则 P 递归地向叶子节点生成终结符 $x \in \Sigma$，就可以得到一个递归下降的生成器。在每一个生成函数中选择不同的递归函数调用路线可以视作在语言 $L(G)$ 中采样，直到达到叶子节点（EOF 等符号也视作叶子节点），语言生成完毕。

4.5　异步、协程和生成器函数

自顶向下的解析器有一个典型问题：通常在解析字符串 $s \in \Sigma^*$ 之前，字符串 s 应该已经完全加载到内存中。这带来的问题是如果在一开始就 $s \notin L(G)$ 或数据在信息传输过程中损坏，解析器仍然要等待整个信息传输完毕，然后再进行解析，当然最终结果一定是解析失败。如果传输了一个 1024GiB 的超大 JSON 数据文件，然后解析失败了，这显然带来了解析之前的大量计算损耗。

对于大语言模型也是同样的道理，如果语言模型输出了3000个Token的序列，然后再由后面的程序解析时发现DSL有误，信息就无法被识别了，那么大量的神经网络计算就“浪费”掉了。解决这个问题的一个方案是采用异步和协程。

异步是指多个操作可以同时发生，而不必等待彼此完成（阻塞）[47]。这可以通过多种方式实现，例如多线程、多进程或事件驱动编程。协程是一种异步的计算机程序组件，是可以暂停和恢复执行的函数。协程通常用于实现异步编程，因为它允许在一个线程中执行多个操作，而不会阻塞其他操作。

生成器（也被叫作“半协程”[48]）是协程的一个子集。尽管两者都能够通过多次 yield 表达式暂停执行并允许在多个入口点重新进入，但它们的主要区别在于协程能够控制让出控制权后程序继续执行的位置，而生成器则不能。生成器只能将控制权转移回其调用者[49]。换言之，由于生成器函数主要用于简化迭代器的编写，其 yield 语句并不是用于跳转到另一个协程，而是将值传回调用它的父级例程。

各种高级语言所支持的生成器函数是实现协程的一种方式，它可以生成一个值的序列。生成器函数使用 yield 关键字来暂停当执行（同时会保留当前执行的上下文）并产生一个值。以 TypeScript 为例，生成器 `G` 可以通过 `G.next()` 函数来恢复执行，`G.return()` 函数来结束执行，以及 `G.throw()` 函数来抛出异常。

正如图 4-9 中所示的 `async` 和 `await` 关键字，`async` 代表一个函数是异步的（那么它的调用会返回一个生成器对象），`await` 代表等待一个异步操作完成。这种异步的生成器函数可以用来解析和生成 DSL，因为它可以在解析或生成过程中暂停和恢复执行，而不会阻塞其他操作。

例如当我们想要解析一个形如 `-123.456e-7` 的数字时，我们可以用异步生成器函数来实现，如图 4-9 所示。首先解析符号部分 `'-'`，其次解析整数部分 `'123'`，之后遇到小数点 `'.'` 则解析小数部分 `'456'`，最后遇到 `'e'` 则解析指数的符号部分 `'-'` 和指数的整数部分 `'7'`。在每一步解析完成后，我们可以暂停解析器，等待其他操作完成，然后再恢复解析器继续解析。

如果采用异步的递归下降解析器，同时采用异步的字符串读取器（其构建了一个缓冲区用于加载字符串），那么两个装置就能互相配合形成一种基于协程的装置。这样的解析器可以一边加载字符串，一边解析 DSL 为指定的数据结构，并且可以在解析器遇到错误时立马停止加载和解析，节约了计算资源。

4.6 用协程产生领域特定语言

协程的实现不依赖任何编程语言的特性，虽然有些编程语言不支持 yield 关键字，但是任何图灵机都可以等价地实现 yield，因为 yield 可以被转换为“switch-case”和上下文状态的存储。

实际上，任何 `async` 和 `await` 都可以被等价地转换为生成器函数和 `yield` 。这也就是说，异步是可以基于协程的。因此，异步的递归下降解析器也可以被等价地转换为生成器函数的递归下降解析器。

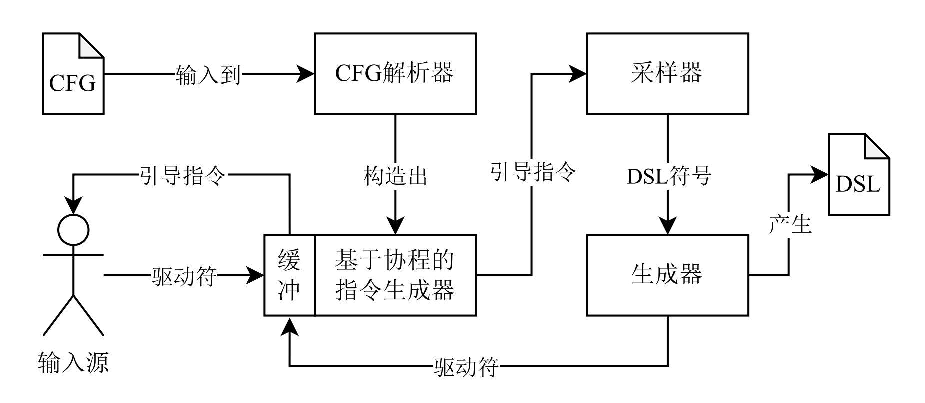


图 4-10　一种通过基于异步的 DSL 产生装置

本文设计的一种协程装置如图 4-10 所示。它接受用户或代理程序的输入流并被驱动，通过异步和协程在装置内维护了 DSL 的解析和生成状态。每一次对装置的驱动被称为一次询问（询问可以携带信息，也可以是空询问），每一次询问装置都会返回一个引导指令。引导指令可以携带期待生成的符号集描述、精确的错误信息和恢复信息、读写操作信息和 DSL 前缀的信息等，用户和或代理程序接受引导指令继续驱动此装置，直到输入 EOF。这个过程中，装置异步解析输入流为约定的数据结构，确保被应用程序正确处理。

此装置接受开发者预定义的 CFG，通过此装置的 CFG 解析器构造出一个异步引导指令生成器。此生成器经过缓冲区接受输入源的驱动符，产生引导指令 (Instruction DSL) 反馈给输入源。输入源也可以是采样器和生成器。采样器可以通过随机、自定义逻辑或神经网络等方式将引导指令转换为一个 DSL 符号，交给生成器产生 DSL，同时驱动异步指令生成器继续产生引导指令，直到得到 EOF 指令，领域特定语言的生成任务就完成了。

此装置的关键点在于采用“异步机制”和“生成指令”。具体来说，此装置第一个技术关键点，异步语言引导和生成机制采用了 yield* (yield from) 。此装置的第二个技术关键点，用一个异步的生成装置产生用于产生 DSL 的指令。

```
let input = yield new Instruction(...args);
```

图 4-11　产生用于生成 DSL 的指令的语句（伪代码）

图 4-11 描述了此装置通过 yield 产生 Instruction DSL 用于指导 DSL 验证、解析和生成，其中 args 是这条指令携带的信息，input 是程序 B（DSL 验证、解析和生成器）收到指令后向程序 A 传递的信息。yield 允许程序 A（DSL 解析程序）立即返回一个指令生成器 (Generator) 给程序 B 。

```
let instruction = Generator.next(input);
```

图 4-12　产生用于生成 DSL 的指令的语句（伪代码）

如图 4-12，接着程序 A 在运行中途暂停。然后等待程序 B 用询问触发指令生成器的 next 方法，恢复程序 A 运行，同时程序 A 拿到询问，程序 A 运行到 yield 处产生引导指令，程序 B 通过 next 方法的返回值得到程序 A 的引导指令。重复这个过程，直到程序 B 不断触发程序 A 运行到 DSL 分析相关算法结束。

yield* 指的是将 DSL 引导指令的产生委托给另一个程序。yield* 同样属于部分编程语言的高级特性。通过 yield* 使得 DSL 的形式 CFG 可以被模块化表达，且可以采用递归的算法处理或生成 DSL。把此装置启动的顶层称做 top 函数，若此装置接受 JSON 的 CFG，则 `let top = yield* json` 表示装置顶层产生 JSON 。

```
let json = yield* select(
    number,
    string,
    object,
    array,
    false,
    true,
    null
)
```

图 4-13　产生用于生成 JSON 的 CFG 语句（伪代码）

而图 4-13 表示 JSON 由数字或字符串或列表或对象或真或假或空中的一个构成。同理，具体 number, string, array… 的文法是怎样的也可以根据 CFG 来确定。上图表示中 false, true, null 是终结符，因为其由确定的字符串 “false” , “true” 和 “null” 构成，而 object, array 产生过程可以产生 number, string, false, true, null 甚至其自身 object 或 array 形成递归定义。由于 yield* 机制可以委托异步任务，因此可以实现递归定义。此装置可以根据此 CFG 产生等价的异步程序，其他任何 CFG 也是这样。

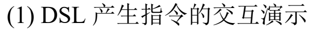
此装置的另一个关键点在于异步产生引导指令 (Instruction DSL)，引导指令是验证，解析和生成 DSL 的信息。也能够根据异步输入提供精确的错误信息、错误恢复信息等。接下来将特定任务上的最小信息单位称作 token，例如可以是 bit, byte, char, number 等，尤其可以是字符或单词。具体来说，由如表 4-2 所示的引导指令包含控制指令、限制指令、建议指令和符号指令。

表 4-2　用于生成 DSL 的指令举例

指令类型	指令举例
控制指令	仅查看接下来 N 个 token 的指令、读入接下来 N 个 token 的指令、撤销前 N 次读入的指令、产生某些 token 的指令、EOF 指令等。
限制指令	限制下一次读入具体某个 token 的指令、限制下一次读入 token 仅有某些具体可选的指令、限制下一次读入 token 应该在某个范围的指令、限制下一次读入 token 不能是某些具体的指令、限制下一次读入 token 不应该在某个范围的指令等。
建议指令	已读入的倒数第 N 个 token 不符合语法的指令、建议下一次读入 token 是具体哪些的指令、建议下一次读入 token 处在哪个范围的指令等。
符号指令	进入某非终结符 $m \in N$ 的指令，离开某非终结符的指令、完成生成某终结符 $x \in \Sigma$ 的指令等。

4.7　产生语言的协程装置演示




(1) DSL 产生指令的交互演示



(2) 在一个简单的 DSL 上演示

图 4-14　采用 Section 4.6 中装置的一个交互 Demo

图 4-14 是采用 Section 4.6 中装置的一个交互 Demo 预览界面。这个 Demo 提供了一个人机交互界面用于人工操作此装置。为了易于理解，此 Demo 配置了一个简单语法的 CFG，即形如 `let abc = 123;` 的简单文法。交互主要包含三部分：符号

输入框、控制按钮、DSL 生成预览和引导指令生成预览。其中符号输入框是一个输入源，可以采集键盘输入的符号，然后输入到本装置的缓冲区。控制按钮可以输入符号输入框无法输入的控制符，例如下一步 (next) 、空 (null) 、结束符 (stop) 。点击自动生成按钮，本装置能够生成下一个符号一次，若自动生成开关打开，则会一直生成到 DSL 结束符，无需其他输入。刷新页面按钮可以重置本装置和此交互界面。DSL 生成预览区域显示了本装置从输入源采集到的合法的 DSL 前缀。引导指令生成预览区域显示了本装置根据输入源（输入框或采样器）生成的引导 DSL 生成指令，此指令能够进一步指导生成装置在 DSL 预览区域生成新的合法 DSL 前缀。

一个异步、协程、高鲁棒性的 DSL 产生和解析装置可以用于信息传输、集成开发、教育软件、数据挖掘等领域。表 4-3 介绍了 Section 4.6 中装置的用途。

表 4-3　基于协程的 DSL 产生装置可能的用途

领域	可能具有的用途介绍
信息传输	此装置能够以更高的鲁棒性异步解析 DSL 为数据结构并驱动其他程序运行，无需等待整个传输完成之后再进行解析，也不会因为传入消息的部分 DSL 结构出错和信息丢失而导致程序崩溃。
集成开发	此装置将能够以更高的效率响应用户的输入，并实时反馈 DSL 生成的引导信息或出错信息，以帮助开发者输入。
教育软件	此装置能够提供精确的拼写建议、错误反馈和修复建议。因此适合作为 DSL 的学习向导，帮助学生快速掌握 DSL 的文法。
数据挖掘	此装置能够从大量文本中逐一扫描字符，并提取符合 CFG 的字符串，从大量数据（例如大量文本）中找到目标 DSL

4.8　本章小结

首先，本章介绍了形式语言的层次和基本概念，给出了上下文无关语言与产生式的关系，然后介绍了形如 BNF 的元语言来表示语法，并列举了常见的领域特定语言案例和它们的简要语法规则。之后，本章介绍了递归下降的语法解析器和生成器，提出了常见解析器因不支持异步而可能造成计算资源浪费的问题。本章还介绍了异步、协程和生成器函数之间的关系（简记为 异步 ⊂ 生成器函数 ⊂ 协程 ），并给出了一个基于 yield* 的 DSL 生成装置来解决上述问题。最后，本章介绍了一个基于协程的 DSL 生成装置的交互演示和可能的用途。

5　生成开发者预期的 DSL 通过语言模型采样

有了基于协程的语言产生或解析装置后，还需要一个输入源。输入源可以是人类，也可以是特定的计算机程序。本章将重点介绍通过基于自回归模型的语言模型作为输入源，并用约束装置限制语言模型的采样，以生成开发者预期的 DSL。

5.1　自回归语言模型

在统计学、计量经济学和信号处理等领域中，自回归模型 (Autoregressive model) 是一种描述随机过程的模型。它可以用于描述某些随时间变化的自然过程。自回归模型 $AR(p)$ 可以被公式 (5-1) 描述，其中 X_t 是时间 t 的随机变量，p 是模型的阶数，φ_i 是模型的系数，ε_t 是时间 t 的随机误差。

$$X_t = \sum_{i=1}^{p} \varphi_i X_{t-i} + \varepsilon_t \tag{5-1}$$

自 2017 年著名的 Transformer 模型诞生开始，它为各个领域提供了灵感和启发。不同领域的研究人员基于 Transformer 研究，衍生出了一系列模型，例如预测时间序列，搜索推荐领域的点击率、用户留存率预测等模型，而著名的 Transformer 正是一种基于自回归的模型[50]。2022 年底，OpenAI 发布的风靡全球的 ChatGPT 正是自回归模型的著名代表，是一种从左往右学习的模型。

自回归模型利用上文，通过估计字母表 Σ 中终结符的概率分布，预测下一个终结符。若有字母表 Σ，给定输入的串 $x = (x_1 \cdot x_2 \cdot \ldots \cdot x_n) \in \Sigma^*$，自回归模型可以计算出下一个终结符 x_{n+1} 的概率分布 $p(x_{n+1})$ 如公式 (5-2)，那么这种自回归模型就是自回的归语言模型，可以用于生成自然语言或 DSL 等诸多类型的任务。

$$\begin{aligned} p(x) &= \prod_{i=1}^{n} p(x_i \mid x_1 \cdot x_2 \cdot \ldots \cdot x_{i-1}) \\ p(x_{n+1}) &= p(x_{n+1} \mid x_1 \cdot x_2 \cdot \ldots \cdot x_n) \end{aligned} \tag{5-2}$$

5.2　语言模型生成 DSL 的基本思路

接下来以 JSON 文法为例，举例语言生成和修复的基本逻辑。若输入源已经输入了片段 `{ "key"`，那么 Section 4.6 所述的协程装置可以根据 JSON 的 CFG 知道

下一个字符必须是冒号。为什么是冒号？这需要查看 JSON 官方[1]以 McKeeman Form 约定的 JSON 语法。如图 4-4 所示的 object, members 和 member 的定义。

```
object  := '{' ws '}'
         | '{' members '}'
members := member
         | member ',' members
member  := ws string ws ':' element
string  := '"' characters '"'
```

图 5-1　JSON 文法中 object, members 和 member 的定义

我们始终忽略空白字符 (ws)，上文 `{ "key"` 的产生路径是 object → members → member，且 member 已经产生到 string 的末尾了。要产生 member，下一个字符只能是冒号。输入源的下一个字符如果不是冒号，就必然不符合 JSON 语法。

同理，上文 `{ "key": 1,` 的下一个字符只能是双引号，因为根据 CFG，末尾的逗号意味着这个 object 的 members 还没结束，逗号后面必须是新的 members，而 members 以 string 开头，string 的开头必须是双引号。

以 JSON 为例子，图 5-2 中的大语言模型表示 JSON 的第一个字符只能是集合 `['{', '[', '"', '0', /1-9/, '-', 't', 'f', 'n']` 中的一个字符，若要产生 JSON，生成器只能从这个集合中选择一个字符作为 JSON 字符串的开头。

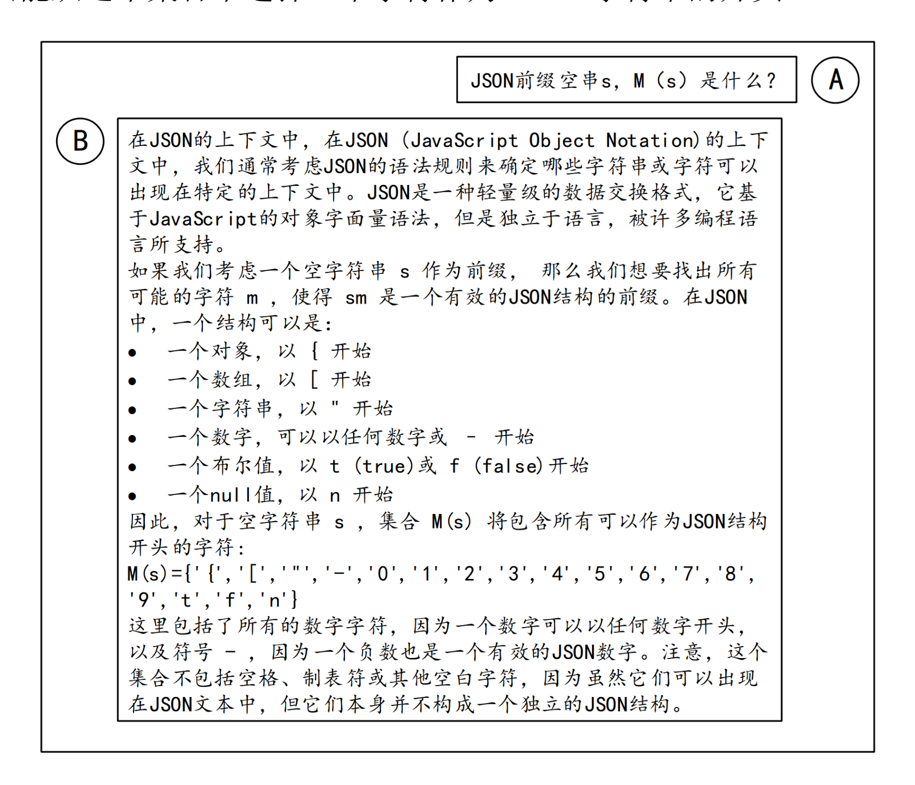


图 5-2　向 LLMs 询问关于 JSON 可能的开头字符集

[1]https://www.json.org/json-en.html

当输入了正确的字符，例如输入了字符 '-' 那么下一个字符只能是 '0' 或 /1-9/（这表示了只能匹配 1, 2, 3, ..., 9 其中的一个字符的 JavaScript 正则表达式），因为上文 '-' 导致接下来要构成 JSON 只能继续构成 number 并以此类推。

```
> JSON.parse(`{ "key": 0`)
✖ ▶Uncaught SyntaxError: Expected ',' or '}' after property  VM234:1
value in JSON at position 10 (line 1 column 11)
    at JSON.parse (<anonymous>)
    at <anonymous>:1:6
```

图 5-3　Microsoft Edge 浏览器无法给出足够精确的提示

以 Microsoft Edge 浏览器的 V8 JavaScript 运行时举例，如图 5-3 所示，Edge 显示期待的字符仅有 ',' 和 '}' 而缺少了 '.', 'E', 'e' 。因此它不能给出完整和精确的错误和恢复信息。其他 Parser Generator（例如 Antlr、Lark 等）的 Parser 也是类似的情况，无法做到给出错误范围，错误恢复建议精确到足以生成 DSL 的程度。

输入 JSON 上文

JSON 上文

```
1  { "key": 0
```

解析上文

期待的字符/正则

在解析到第 10 个字符时， APNTs 如下：

	期待值 (Char/RegExp)	类型
1	.	Char
2	E	Char
3	e	Char
4	,	Char
5	}	Char
	期待值 (Char/RegExp)	类型

图 5-4　计算所有可能的下一个字符集的装置演示

图 5-4 所示的是本研究的装置在 JSON 作为 CFG 的一个可视化例子，当解析 `{ "key": 0` 时，此装置给出了更加完整和精确的信息以供 DSL 的产生。

图 5-5 给出了一个示意图，它一定程度上表达了 JSON CFG 中的符号构成的产生关系。可以发现这个图的根节点是 json 节点，并且图中明显有众多环。如果让语言模型产生 JSON，就相当于它在这样一个复杂的、迷宫一般的图上行走、选择路线和识别终点。这可以作为语言模型如果不依赖更好的解码装置，就难以生成更长的、合法的 JSON 字符串的原因。

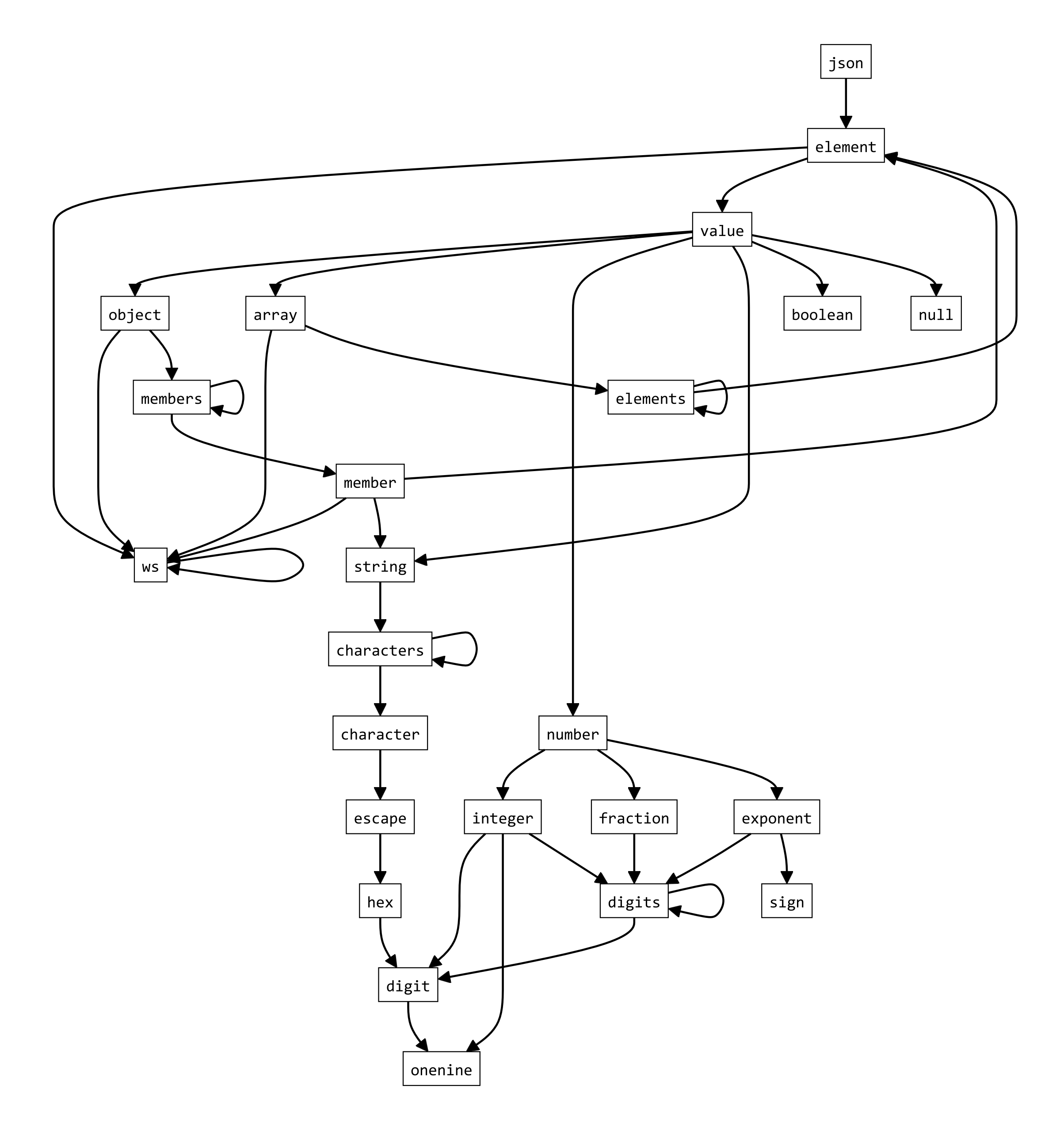


图 5-5　JSON CFG 的一个有环示意图

若定义字母表 Σ，上下文无关文法 G 和语言 $L(G)$，将公式 (5-3) 中的 $P(G)$ 称为语言 $L(G)$ 上的前缀语言。

$$P(G) = \{x \in \Sigma^* | \exists y \in \Sigma^* : xy \in L(G)\} \tag{5-3}$$

$$M_P(s) = \{m \in \Sigma | sm \in P(G)\} \tag{5-4}$$

若有一个装置，能够在串 $s \in P(G)$ 的基础上给出下一个合法的词元集合 $M_P(s)$，然后语言模型根据公式 (5-4) 中的 $M_P(s)$ 上的概率分布 $p(x_{n+1})$ 采样，就可以生成新的 $s' \in P(G)$，直到装置从语法 G 中产生结束符，模型就以生成到 $s' \in L(G)$ 而停止继续生成。这就是从语言模型生成开发者预期的 DSL 的方法。

5.3 自回归语言模型的约束解码器

自从来自 ElementAI 的 Torsten Scholak 等人于 2021 年提出了增量解析来约束语言模型的自回归解码 (PICARD)[4] 之后， MIT 的研究人员采用基于 Earley 解析的算法来约束大模型生成[6]，其基本思路是构造了一个能够根据语言模型产生的字串 $\hat{y}$ 下计算得出语言 $L(G)$ 的一个前缀 $y_{\text{prefix}} \in P(G)$ 和下一个合法的词元集合 $M_P(y_{\text{prefix}})$ 的装置，然后再经由语言模型得出一个具体的 $\omega^* \in M_P(y_{\text{prefix}})$，将其添加到 y_{prefix} 末尾，得到新的 DSL 前缀，直到 $\hat{y} \in L(G)$ ，如图 5-6 所示。

CONSTRAINED-GENERATION $(x \in \Sigma^*, G)$:
$\hat{y} \leftarrow \varepsilon$ ▷ 初始化空串 ε
while True:
  $\bar{y} \leftarrow \text{decode}(P_{\text{LLM}}(\cdot\,|x, G, \hat{y}, ...))$
  $\hat{y} \leftarrow \hat{y} \cdot \bar{y}$ ▷ 连接串并更新 $\hat{y}$
  if $\hat{y} \in L(G)$**:** ▷ 尝试验证串 $\hat{y} \in L(G)$
    return $\hat{y}$ ▷ 返回预期的 DSL
  else:
    $y_{\text{prefix}}, M_P(y_{\text{prefix}}) \leftarrow \text{Generator}(\hat{y}, G)$
    $\omega^* \leftarrow \arg\max P_{\text{LLM}}(\omega|\omega \in M_P(y_{\text{prefix}}), y_{\text{prefix}}, ...)$
    $\hat{y} \leftarrow y_{\text{prefix}} \cdot \omega^*$ ▷ 更新 $\hat{y} \in P(G)$

图 5-6 约束大模型生成 DSL 的算法[6]

现有工业实践和学术研究中，尚未采用协程来实现图 5-6 中的 Generator 装置。且容易知道，上述算法在每一次自回归当中都需要验证一次 $\hat{y} \in L(G)$，就算是验证的时间复杂度随 $|\hat{y}|$ 的长度 n 是 $O(n)$（这是最好的情况，也是复杂度的下限），那么生成长度为 $\hat{y}$ 的复杂度至少为 $O(n^2) = \sum_{i=1}^{n} O(n)$ 。

本研究采用的协程因为可以在解析或生成过程中暂停以及恢复，因此自然地保留了 DSL 解析的上下文状态，不需要在每一次验证或生成新词元时都从头扫描一遍字符串，这为降低约束解码的时间复杂度提供了技术支持。

5.4 用 DSL 生成器和约束装置采样

那么如何具体设计一个采用协程的 DSL 生成器，并将它作为语言模型的约束装置来采样开发者预期的领域特定语言呢？本文提出了 YieldLang，一个采用 Python 元编程技巧实现的基于协程的 DSL 生成器。YieldLang 的设计思路是将形式语法的符号统一处理为可迭代产生字符串的装置，同时跟踪符号进出的调用栈，以便生成 DSL 的过程中能够提供精确的错误信息和恢复信息。此外，当 YieldLang 的生成器

装载语言模型采样器时，它的组合子函数能够调用大模型来从有限的可能中采样路径并递归下降的生成 DSL。

5.4.1 DSL生成器的一个具体实现

本文采用 Python 实现了基于协程的 DSL 生成器，其关键点在于文本生成器类 `TextGenerator` 中的 `__flatten` 递归函数，如图 5-7 所示。它使用元编程的方法将形式文法里的符号（包括终结符和非终结符）Symbol 通过委托 (yield from) 等方式统一地转换为产生字符串的可迭代装置。Symbol 可以是可转换为字符串的 Strable、Unicode 以外的特定 Token、空（包括 None、空串、代表空的 Token 等）、可迭代产生上述类型的 Iterable 以及可以产生上述类型的 Lambda 函数。

```python
def __flatten(self, symbol: Symbol) -> Iterable[str]:
    if callable(symbol):
        symbol = symbol()
    if symbol is None:
        yield EmptyString
    elif isinstance(symbol, (str, int, float, bool)):
        yield str(symbol)
    elif isinstance(symbol, Token):
        yield from self.__process_token(symbol)
    elif isinstance(symbol, SamplerCallData):
        yield from self.__process_sampler(symbol)
    elif iterable(symbol):
        yield from self.__process_iterable(symbol)
    else:
        raise ValueError(f'Invalid symbol: {symbol}')
```

图 5-7　YieldLang 将符号统一处理为可迭代产生字符串的装置

如果这个装置仅仅能够生成合法 DSL，似乎还是有点扫兴。幸好，本文设计了 `track_symbol` 函数，这是一个用于记录符号进出情况的装饰器，如图 5-8 所示。它把递归下降的函数调用参数都记录到了一个调用栈当中。在生成器外部，总是能够获取调用栈的栈顶就可以获取符号名等信息。当调用栈发生变化时，就可以得知 DSL 的生成情况，当然也可以跟踪整个调用栈，并把它恢复成一颗语法树。

```python
def track_symbol(fn: T) -> T:
    @functools.wraps(fn)
    def wrapper(*args, **kwargs):
        self = args[0] if args else None
        if isinstance(self, TextGenerator):
            self.sampler.call_stack.append((fn, args, kwargs))
            yield from fn(*args, **kwargs)
            self.sampler.call_stack.pop()
        else:
            fn_name = fn.__name__
            class_name = TextGenerator.__name__
            raise ValueError(f'{fn_name} is not a method of {class_name}')

    return wrapper
```

图 5-8　YieldLang 跟踪符号进出的装饰器函数

此外，此装置的文本生成器类 `TextGenerator` 中还统一定义了 Top 函数接口，代表形式语言中的开始字符 $S \in N$，它是生成器产生 DSL 的唯一入口。通过 Python 的元编程技巧，例如装饰器 (Decorator)、组合子 (Combinator)、函子 (Functor) 等方式，可以实现一套用于产生 DSL 的应用程序开发框架。基于此框架，开发者能够在 Python 的语法基础上书写元语言，此框架被本文称作 YieldLang。

5.4.2 YieldLang的一些DSL生成器

```
class PairsGenerator(TextGenerator):
    def top(self):
        yield self.pairs

    def pairs(self):
        yield select(
            (self.pair),
            (self.pair, self.pairs)
        )

    def pair(self):
        yield optional(
            '(', self.pairs, ')'
        )
```

图 5-9 YieldLang 下的一个合法括号对生成器

如图 5-9 所示，这是一个生成合法括号对的生成器。它的 `top` 函数是生成器的入口，它调用了 `pairs` 函数，而 `pairs` 函数又调用了 `pair` 函数。`pair` 函数是一个产生括号对的函数，它产生了两种可能的括号对：空串和括号对。这个生成器可以生成任意数量的合法括号对，例如 (), (()), (())(), (()())(), ... 等等。其中 select 组合子函数用于选择一个采样路径，optional 函数等价于如图 5-10 所示的代码。

```
def optional(self, *args: Symbol):
    yield select(
        (''),
        args
    )
```

图 5-10 YieldLang 中 optional 组合子的定义

另一个较为复杂的例子是生成简单的 Mermaid 流程图，其生成器代码如图 5-11 所示。其中的 Python 表达式 (`yield self.graph_name`) 会得到一个符号产生的字符串，然后根据这个字符串的内容，选择产生 `flowchart`（作为一个简单的例子，这里不产生其他类型的 Mermaid 图）。值得一提的是 (`yield A`) 等价于 `yield (yield A)`。`join` 方法用于在最外层符号序列中的每相邻两个符号之间插入一个特定的符号。`repeat` 方法用于重复产生一个符号指定次数。

```python
class MermaidGenerator(TextGenerator):
    def top(self):
        yield self.mermaid

    def mermaid(self):
        match (yield self.graph_name):
            case 'flowchart':
                yield self.flowchart

    def graph_name(self):
        yield select('flowchart')

    def flowchart(self):
        yield (' ', self.flowchart_type, '\n')
        yield join('\n', self.flowchart_rules)

    def flowchart_type(self):
        yield select('TD', 'LR')

    def flowchart_rules(self):
        rand_times = randint(10, 20)
        single_line = (' ' * 4, self.flowchart_rule)
        yield from repeat(single_line, rand_times)

    def flowchart_rule(self):
        yield self.node
        yield ' --> '
        yield self.node

    def node(self):
        yield select(*range(1, 10))
```

图 5-11　YieldLang 下的一个简单流程图生成器

如果采用随机的 `class RandomSampler(Sampler)` 采样器（它在每一个交叉路口随机选择一条路径递归下降的采样），那么图 5-11 的基于 YieldLang 的生成器可以产生形如图 5-12 中所示的随机节点和随机有向边的流程图。

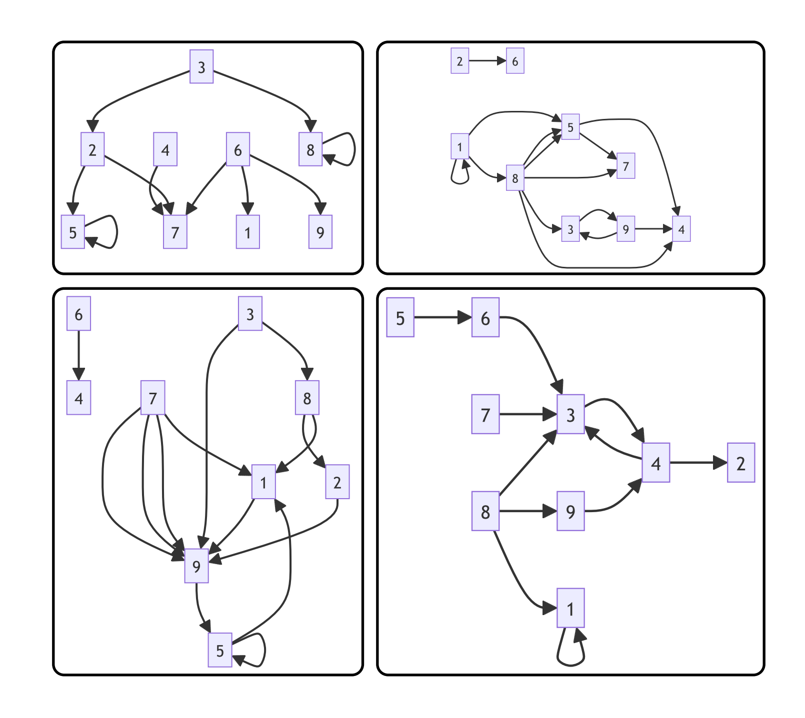


图 5-12　Mermaid Generator 随机生成的四个流程图

```python
def print_sample_rules():
    rules: set[str] = set()
    for _ in range(sample_times):
        sampler = RandomSampler()
        for _ in JSONGenerator(sampler):
            last_symbol = None
            for symbol_name in sampler.symbol_names():
                if last_symbol and symbol_name:
                    rule = f'{last_symbol} --> {symbol_name}'
                    if rule not in rules:
                        rules.add(rule)
                        print(rule)
                last_symbol = symbol_name
```

图 5-13　从已有的生成器中随机采样产生规则

通过随机采样和 `track_symbol` 函数，还可以通过图 5-13 所示的 `print_sample_rules` 方法随机采样此生成器下的 CFG 中的产生规则。这个方法可以用于生成 DSL 的语法树或产生规则图（采用 Mermaid 来解析和渲染这种图）。图 5-5 就是采用这个方法生成的 JSON 语法的产生规则图。

JSON 作为工业界非常常用的一种 DSL，本文提出的 YieldLang 显然也能够用于定义它的语法和构造对应的生成器装置。图 5-14 就给出了完整的 JSON 语法的生成器代码，它的 CFG 是按照 JSON 官方的 McKeeman Form 所书写的。

JSONGenerator 中的 accept 函数用于定义这个符号所能接受的字符集（或者说是能够产生的终结符集合 $\mathrm{accept}(\ldots) \subseteq \Sigma$ ），其中 invalids 参数用于定义这个符号所不能接受的字符集合，range 参数用于定义这个符号所能接受的字符范围。有了 JSONGenerator 后，通过随机采样就能够生成任意长度的随机 JSON 字符串。

表 5-1　随机采样生成的 JSON 字符串样例

JSON 样例	对象类型		字符串长度
`null`	`none`	(null)	4
`"⎍"`	`string`	(string)	5
`false`	`boolean`	(boolean)	5
`-39.01`	`float`	(number)	6
`-96008`	`integer`	(number)	6
`"ꙍ責"`	`string`	(string)	8
`"/鴶?\""`	`string`	(string)	9
`[[[true]]]`	`array`	(array)	10
`{"껇ɿ": []}`	`dictionary`	(object)	13

表 5-1 列举了随机采样生成的 JSON 字符串的不同样例，它们的类型和长度各不相同。需要注意的是，随机 JSON 字符串样例是没有任何语义的，这里仅表示其语法结构，其中的“乱码”是由于随机从 Unicode 字符集中挑选字符而生成的。

```python
class JSONGenerator(TextGenerator):
    def top(self):
        yield self.json
    def json(self):
        yield self.element
    def object(self):
        yield select(
            ('{', self.ws, '}'),
            ('{', self.members, '}')
        )
    def members(self):
        yield select(
            (self.member),
            (self.member, ',', self.members)
        )
    def member(self):
        yield (self.ws, self.string, self.ws, ':', self.element)
    def array(self):
        yield select(
            ('[', self.ws, ']'),
            ('[', self.elements, ']')
        )
    def elements(self):
        yield select(
            (self.element),
            (self.element, ',',  self.elements)
        )
    def string(self):
        yield ('"', self.characters, '"')
    def characters(self):
        yield optional(self.character, self.characters)
    def character(self):
        yield select(
            accept(range=('\u0020', '\uffff'), invalids=('"', '\\')),
            ('\\', self.escape)
        )
    def escape(self):
        yield select(
            *'\\"/bfnrt',
            ('u', repeat(self.hex, 4))
        )
    def hex(self):
        yield select(
            self.digit,
            select(*'ABCDEF'),
            select(*'abcdef')
        )
    def digit(self):
        yield select('0', self.onenine)
    def onenine(self):
        yield select(*'123456789')
    def number(self):
        yield (self.integer, self.fraction, self.exponent)
    def integer(self):
        yield select(
            (self.digit),
            (self.onenine, self.digits),
            ('-', self.digit),
            ('-', self.onenine, self.digits)
        )
    def digits(self):
        yield select(
            (self.digit),
            (self.digit, self.digits)
        )
    def fraction(self):
        yield optional('.', self.digits)
    def exponent(self):
        yield optional(select(
            ('E', self.sign, self.digits),
            ('e', self.sign, self.digits)
        ))
    def sign(self):
        yield optional(select('+', '-'))
    def boolean(self):
        yield select('true', 'false')
    def null(self):
        yield 'null'
    def value(self):
        yield select(
            self.object,
            self.array,
            self.string,
            self.number,
            self.boolean,
            self.null
        )
    def element(self):
        yield (self.ws, self.value, self.ws)
    def ws(self):
        yield optional(select(
            ('\u0020', self.ws),
            ('\u000A', self.ws),
            ('\u000D', self.ws),
            ('\u0009', self.ws)
        ))
```

图 5-14　YieldLang 下的一个 JSON 字符串生成器

5.4.3 如何让语言模型选择采样路径

仅仅有随机采样器肯定是不够的，为了利用语言模型的能力生成真正有语义的 DSL，需要让语言模型选择采样路径。然而，语言模型的神经网络输出层能够代表下一个词在词汇表 Σ 上的概率分布 $p(x_{n+1})$ ，但是却不能直接给出非终结符的概率分布。这导致我们并不能直接的让语言模型在规则中采样路径，而是首先需要找到"分岔路口"中每一个路口对应的非终结符 $A \in N$ 的 First 集合（如果某个路径直接到达终结符，那么就可以直接选取它）。

在形式语言和编译原理中，First 集合和是用于预测上下文无关文法 (CFG) 中某个非终结符的下一个输入符号的重要概念。First 集合包含一个非终结符可能产生的第一个符号的集合，它用于预测某个非终结符是否可以推导出某个符号。

对于一个非终结符 A，其 First 集合 $\text{First}(A)$ 由以下三条规则定义：

(1) 如果 A 可以直接推导出空串 ε ，则 $\varepsilon \in \text{First}(A)$。

(2) 如果 A 可以推导出以某个终结符 $a \in \Sigma$ 开始的串，则 $a \in \text{First}(A)$。

(3) 如果 A 可以推导出以非终结符 B 开始的字符串，则 $\text{First}(B) \subseteq \text{First}(A)$。

显然根据 First 集合的三条规则可以设计一个迭代产生式的算法来计算得出 First 集合。这个算法的基本思路是首先初始化 $\text{First}(A) = \emptyset$ ，然后不断迭代直到 $\text{First}(A)$ 不再增大。在每一次迭代中，对于P 中每一个产生式 $A \to \alpha$ ，如果 α 是终结符或者空串，那么就把 α 加入到 $\text{First}(A)$ 中。如果 α 是非终结符，那么就把 $\text{First}(\alpha)$ 中的所有元素加入到 $\text{First}(A)$ 中。这个算法的时间复杂度是 $O(|N| * |P|)$，其中 $|N|$ 是非终结符的数量，$|P|$ 是产生式的数量。对于四元组 $G = (N, \Sigma, P, S)$ 所定义的上下文无关文法，可以由图 5-15 所示的基础算法求得。

```
FIRST (A ∈ N):
s ← ∅
for (A → α) in P:
    if (α → ε):
        s ← s ∪ {ε}
    else if (α = a ∈ Σ):
        s ← s ∪ {a}
    else if (α = B ∈ N):
        s ← s ∪ First(B)
return s
```

图 5-15 一个求 First 集合的算法

但图 5-15 所示算法的问题在于，不能处理存在左递归的情况。左递归是指一个非终结符在产生式的右部以自身开头，例如 $\{A \to Aa, A \to b\}$。在这种情况下求

解 A 的 First 集合时会陷入无限循环。因为 A 的 First 集合可能包含 A，所以求解 A 的 First 集合时需要再次求解 A 的 First 集合，如此形成死循环。此外，如果多个符号之间形成了间接的左递归，同样也会遇到死循环的情况。

为了消除左递归的问题，有两种方法，一是开发人员在书写 DSL 的文法的时候就避免左递归，二是更进一步在生成器构造的时候根据文法规则自动消除左递归。当然，对于具有通用能力的大语言模型，求解不出First 集合也没关系，通过添加指令让语言模型直接选择非终结符（因为非终结符本身也可以转换为字符串，甚至开发者可以描述这个非终结符，从而能够构造出一个或多个属于大语言模型字母表 Σ 的字符串来让大模型采样）。这样语言模型就能够从 DSL 语法中采样路径了。

图 1-3 所示的引导大语言模型生成可解析内容的系统也是利用语言模型作为 DSL 生成器的采样器的流程图。其流程为：i. 输入提示词，语言模型从上文为空开始生成。ii. 异步 DSL 解析和语言生成装置产生引导指令，其被语言模型引导装置转化为下一个合法的词元集合。iii. 语言模型引导装置作用于 NN（神经网络）的输出层，神经网络输出层的 Tensor 通过后处理器得到新的 Tensor，然后通过 softmax 函数转换为语言模型词汇表的概率表示，最后经过采样器得到具体某一个 token。 iv. 语言模型引导装置采用语言模型的词汇表对应的 Tokenizer 将下一个合法的词元集合。 v. 词汇偏置：语言模型引导装置通过为 Logits 和 New Logits 之间装入一个新的约束器 Processor，用以提高概率分布 Tensor 中合法 ID 位置值的大小，降低不合法 ID 位置值的大小。 vi. 经过模型的解码器，模型输出正确的新 token，形成新的 DSL 或 DSL 的前缀。 vii. 通过 send 方法输入 token 到异步装置，得到新的引导指令。 viii. 回到步骤 iii，根据当前任务的上下文继续生成，直到异步装置产生结束指令，开发者预期的 DSL 生成完毕，此 DSL 是一定能够被解析的。

5.5　本章小结

本章先介绍了语言模型常基于的自回归模型，然后以 JSON 语法为例子阐述了 JSON 字符串的产生思路，并指出当前常见解析器的不足。接着，本章给出了前缀语言 $P(G)$ 和装置 $M_P(s)$ 的数学定义。之后本章介绍了自回归语言模型的约束解码策略，然后本文用 Python 语言具体设计了一个 DSL 生成框架，并取名 YieldLang，还举了几个用 YieldLang 作为元语言书写 DSL 的 CFG 的例子。最后，本章介绍了让语言模型在生成器中选择采样路径的方法和 First 集合，最终给出了一个引导大语言模型生成可解析内容的系统的具体流程描述。

6　本研究在语言模型生成 DSL 领域取得优势

本文作者在三个测试集中测试了 GPT-2 和 Gemma 两个模型在温度设置为 0.7 情况下的 DSL 下游任务的能力。JSON Text 任务考虑模型在完整 JSON 文法下生成特定的 JSON 子集下数据结构的准确率。Mermaid 任务考虑模型在 Flowchart 的一个子集 DSL 下生成能被 Mermaid.js[1] 成功渲染为 SVG 格式的矢量图的成功率。Function Call 任务考虑模型生成 Python 函数调用表达式并成功调用的成功率。

在不使用 DSL 的约束装置（基准）和使用本文实现的 Yield Lang 和大语言模型采样装置（本文）两种情况下，后者的性能显著高于前者，就如表 6-1 所示。

表 6-1　本研究在多种 DSL 生成任务上的分数

模型名称	JSON Text		Mermaid		Function Call	
	基准	本文	基准	本文	基准	本文
GPT-2	6.7%	**12.1%**	7.2%	**83.6%**	16.7%	**18.9%**
GPT-2 XL	13.5%	**19.6%**	11.3%	**87.4%**	19.0%	**20.7%**
Gemma-2B	20.4%	**42.1%**	23.2%	**91.1%**	23.7%	**26.4%**
Gemma-7B	29.3%	**49.9%**	34.4%	**97.7%**	28.2%	**31.9%**

其次，由于在生成器产生 DSL 时，某些路径总是能被程序确定的，无需大语言模型进行采样，这节约了大语言模型产生 DSL 所需要的采样次数。经过实验，产生不含有语义的 JSON 字符串，最理想的情况下（当 $\Sigma_{\text{LLM}} \subseteq \text{Unicode}$）大约能将采样次数减少到原来的约 16.5%，随字符串长度的变化如图 6-1 所示。

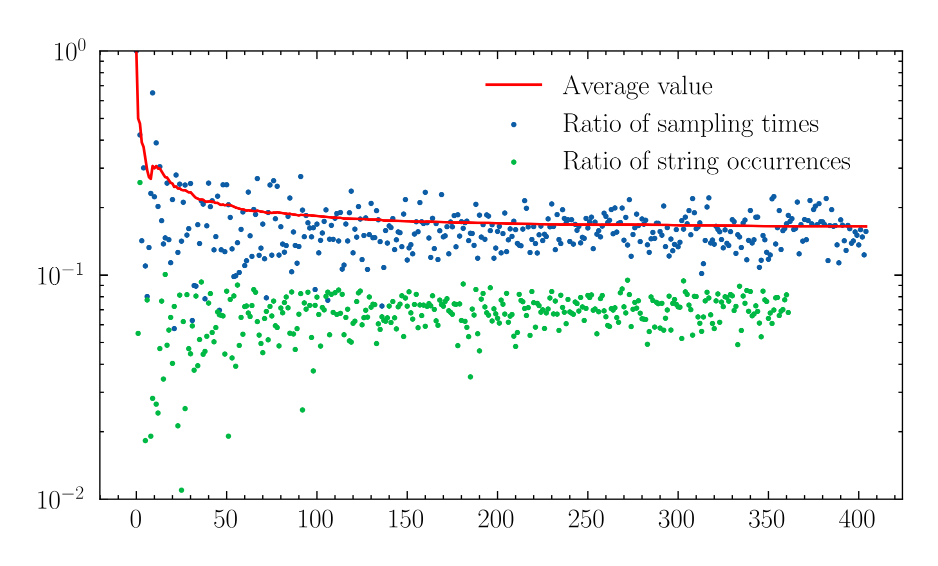


图 6-1　调用 LLMs 的采样次数与生成 JSON 长度的比率

[1]https://mermaid.js.org/

若仅考虑 DSL 能够被 Parser 解析成功，本研究进行了约 2.7 亿次 JSON 生成和用 Python 的 JSON 包解析的测试，生成了平均长度 17.86、最大长度 2136 的 JSON 字符串，其中 100% 的 JSON 字符串都能够被成功解析并还原。

6.1 本研究实现大语言模型 JSON 模式

图 6-2　JSON 模式大语言模型实现的交互界面

JSON 作为典型的 DSL 能够表达常用的数据结构，本研究基于 YieldLang 和 JSON 的语法规则实现了大语言模型的 JSON 模式，如图 6-2 所示。

图 6-2 中的人机交互界面从上到下分别是标题、提示词输入框按钮、提示词输入框、JSON 模式启用开关、生成按钮和生成结果预览。第一个提示词输入框按钮是编辑提示词 Markdown 源码的按钮、第二个是查看输入框使用帮助的按钮。提示词输入框用于输入提供给语言模型的提示词。关闭 JSON 模式，模型将不采用本研究的装置。打开 JSON 模式，将会启用本研究的装置，并配置 JSON 的 CFG。点击生成按钮，语言模型将采用提示词逐个生成 Token 并添加到生成结果预览编辑器的光标末尾，直到模型输出到结束 Token ，大语言模型生成 JSON 完毕。

在工业实践中，基于本研究的装置可以纳入基于 Transformer 的推理框架中，提供一个与 OpenAI 类似的 API，例如 `{"response_format": {"type": "json_object"}}` 表示语言模型应该以 `json_object` 格式返回响应。更进一步，采用提示词工程等技术，还可以在“适当”的时候或“位置”启用 DSL 约束装置，以实现具体的功能。

7 总结与展望

本文研究了大语言模型生成领域特定语言的挑战和方法。首先介绍了 DSL 的广泛应用和研究现状，并指出语言模型生成 DSL 的潜力和局限性。本文分析了 DSL 的典型句法特征，并提出了一种验证语言模型生成 DSL 性能的实验方法。实验结果表明，现有的语言模型在生成 DSL 任务上的性能难以接受。为了解决上述问题，本文提出了一种基于约束解码的 DSL 生成装置，该装置包括异步 DSL 解析语言生成装置和语言模型引导装置。本文还介绍了自回归语言模型的约束解码策略，并基于 Python 的元编程能力设计了 YieldLang 框架，一个易用的 DSL 生成框架。

本研究设计的方案在多个 DSL 生成任务上取得了优势，能够提高大语言模型生成 DSL 的性能。令人喜出望外的是，本研究的装置还有望节约大语言模型的采样次数，这可以减少语言模型生成 DSL 所需要的计算量和时间。本研究还实现了一个大语言模型的 JSON 模式，提供了一个人机交互界面和网络 API，用户可以通过给大语言模型输入提示词来生成特定的 JSON 字符串。最近新出现的 MoonBit 语言已经开始尝试采用 DSL 采样器来“适配”大语言模型[1]，本研究提出的基于协程的方法有望改进现有语言模型的 DSL 生成装置和降低开发难度。

实际上，本研究可以被粗略地认为是一种利用“自动补全”的方法来帮助或限制语言模型生成 DSL。约束解码可以形象地理解为“掐着语言模型的脖子说话”。此外，DSL 作为一种统一的、通用的和已经被大规模应用的形式，有望帮助人工智能掌握工具的使用方法并成为 AGI。下面列举了两个可能的应用场景：

(1) 自动补全器：使用语言模型帮助开发人员用更少的步骤输入代码实现需求。如果语言模型能够根据上下文环境构建更具体的 DSL 语法规则，再经过约束解码来帮助开发人员书写代码，那么可能将大大提高开发者的采纳率。

(2) 智能操作员：根据 CFG 产生 100% 正确的函数调用表达式或其他 DSL，使得模型可以根据上下文环境，以更高的成功率调用函数完成某些特定的任务。

最后，本研究还有众多不足，例如 LLMs 在受到更复杂约束的情况下，它的各种性能与条件概率是怎样的？如何将提示词工程技术、In-Context Learning 的方法与 DSL 约束相结合，使得模型能够在更多场景下拥有好的性能？相关方法是否有益于 LLMs 的预训练或微调？这些问题将在未来的研究中进一步探讨。

[1]https://www.moonbitlang.com/blog/moonbit-ai

参考文献

[1] LIN Y, LIN H, XIONG W, et al. Mitigating the Alignment Tax of RLHF[Z]. 2024.

[2] RADFORD A, NARASIMHAN K, SALIMANS T, et al. Improving language understanding by generative pre-training[J]. 2018.

[3] YAO S, ZHAO J, YU D, et al. ReAct: Synergizing Reasoning and Acting in Language Models[Z]. 2023.

[4] SCHOLAK T, SCHUCHER N, BAHDANAU D. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models[C/OL]// Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, 2021: 9895-9901. https://aclanthology.org/2021.emnlp-main.779.

[5] LAGES J. OpenAI JSON Mode vs Functions[Z]. 2024.

[6] WANG B, WANG Z, WANG X, et al. Grammar prompting for domain-specific language generation with large language models[J]. Advances in Neural Information Processing Systems, 2024, 36.

[7] BACCIANELLA S, POWLEY B, TERRY. mangiucugna/json_repair[M/OL]. 2024. https://github.com/mangiucugna/json_repair.

[8] LUNDBERG S, RIBEIRO M T C, EDGAR R, et al. guidance-ai/guidance[M/OL]. 2024. https://github.com/guidance-ai/guidance.

[9] WOLF T, GUGGER S, DEBUT L, et al. huggingface/transformers[M/OL]. 2024. https://github.com/huggingface/transformers.

[10] GENG S, JOSIFOSKI M, PEYRARD M, et al. Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning[Z]. 2024.

[11] KWON W, LI Z, BAUM A, et al. vllm-project/vllm[M/OL]. 2024. https://github.com/vllm-project/vllm.

[12] SHINAN E, MEGAING, CHANICPANIC, et al. lark-parser/lark[M/OL]. 2024. https://github.com/lark-parser/lark.

[13] BRUNSFELD M, HLYNSKYI A, QURESHI A, et al. tree-sitter/tree-sitter: v0.22.2[M/OL]. Zenodo, 2024. https://zenodo.org/doi/10.5281/zenodo.10827268. DOI:10.5281/ZENODO.10827268.

[14] SAMMET J E. Programming languages: History and fundamentals[M]. Prentice-Hall, Inc., 1969.

[15] WEXELBLAT R L. History of programming languages[M]. Academic Press, 1981.

[16] BRAY T. The JavaScript Object Notation (JSON) Data Interchange Format[EB/OL]. RFC Editor, 2014. https://www.rfc-editor.org/info/rfc7159. DOI:10.17487/RFC7159.

[17] ST.LAURENT S, MAKOTO M, KOHN D. XML Media Types[EB/OL]. RFC Editor, 2001. https://www.rfc-editor.org/info/rfc3023. DOI:10.17487/RFC3023.

[18] MASINTER L M, CONNOLLY D W. The 'text/html' Media Type[EB/OL]. RFC Editor, 2000. https://www.rfc-editor.org/info/rfc2854. DOI:10.17487/RFC2854.

[19] SHAFRANOVICH Y. Common Format and MIME Type for Comma-Separated Values (CSV) Files[EB/OL]. RFC Editor, 2005. https://www.rfc-editor.org/info/rfc4180. DOI:10.17487/RFC4180.

[20] SVEIDQVIST K, MERMAID C to. Mermaid: Generate diagrams from markdown-like text[EB/OL]. (2014-12-02). https://mermaid.js.org/.

[21] FREED N, BORENSTEIN D N S. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies[EB/OL]. RFC Editor, 1996. https://www.rfc-editor.org/info/rfc2045. DOI:10.17487/RFC2045.

[22] ION P, MINER R, AUSBROOKS R, et al. Mathematical Markup Language (MathML) Version 3.0[J]. World Wide Web Consortium, 1998.

[23] MÄDJE L. A Programmable Markup Language for Typesetting[D]. 2022.

[24] SINGHAL K, HIETALA K, MARSHALL S, et al. Q# as a Quantum Algorithmic Language[J/OL]. Electronic Proceedings in Theoretical Computer Science, 2023, 394: 170-191. http://dx.doi.org/10.4204/EPTCS.394.10. DOI:10.4204/eptcs.394.10.

[25] WEININGER D. SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules[J]. Journal of chemical information and computer sciences, 1988, 28(1): 31-36.

[26] O'BOYLE N, DALKE A. DeepSMILES: an adaptation of SMILES for use in machine-learning of chemical structures[J]. 2018.

[27] VANDERMEERSCH T, MAYFIELD J, HUTCHISON G, et al. opensmiles/OpenSMILES[M/OL]. 2021. https://github.com/opensmiles/OpenSMILES.
[28] OFER D, BRANDES N, LINIAL M. The language of proteins: NLP, machine learning & protein sequences[J/OL]. Computational and Structural Biotechnology Journal, 2021, 19: 1750-1758. https://www.sciencedirect.com/science/article/pii/S2001037021000945. DOI:https://doi.org/10.1016/j.csbj.2021.03.022.
[29] WANG G, COOK P R, SALAZAR S. ChucK: A Strongly Timed Computer Music Language[J/OL]. Computer Music Journal, 2015, 39(4): 10-29. https://doi.org/10.1162/COMJ/_a/_00324. DOI:10.1162/COMJ_a_00324.
[30] LAZZARINI V, YI S, HEINTZ J, et al. Csound: a sound and music computing system[M]. Springer, 2016.
[31] BOHNACKER H, GROSS B, LAUB J, et al. Generative design: visualize, program, and create with processing[M]. Princeton Architectural Press, 2012.
[32] WOLFRAM S. What is ChatGPT doing ... and why does it work?[EB/OL]. Stephen Wolfram Writings, 2023[2023-02-14]. https://writings.stephenwolfram.com/2023/02/what-is-chatgpt-doing-and-why-does-it-work.
[33] RADFORD A, WU J, CHILD R, et al. Language models are unsupervised multitask learners[J]. OpenAI blog, 2019, 1(8): 9-10.
[34] TEAM G, MESNARD T, HARDIN C, et al. Gemma: Open Models Based on Gemini Research and Technology[J]. arXiv preprint arXiv:2403.08295, 2024.
[35] WANG C, LIU X, AWADALLAH A H. Cost-effective hyperparameter optimization for large language model generation inference[C]//International Conference on Automated Machine Learning. 2023: 21-21.
[36] ACHIAM J, ADLER S, AGARWAL S, et al. GPT-4 technical report[J]. arXiv preprint arXiv:2303.08774, 2023.
[37] TOUVRON H, MARTIN L, STONE K, et al. Llama 2: Open foundation and fine-tuned chat models[J]. arXiv preprint arXiv:2307.09288, 2023.
[38] RENZE M, GUVEN E. The Effect of Sampling Temperature on Problem Solving in Large Language Models[J]. arXiv preprint arXiv:2402.05201, 2024.
[39] NEWYSTATS. Plot of the probability density function of the exponential distribution[Z]. 2019.

[40] CHOMSKY N. Three models for the description of language[J]. IRE Transactions on information theory, 1956, 2(3): 113-124.

[41] MASCARENHAS F, MEDEIROS S, IERUSALIMSCHY R. On the relation between context-free grammars and parsing expression grammars[J/OL]. Science of Computer Programming, 2014, 89: 235-250. https://www.sciencedirect.com/science/article/pii/S0167642314000276. DOI:https://doi.org/10.1016/j.scico.2014.01.012.

[42] KERNIGHAN B W, RITCHIE D M. The C Programming Language[M]. 2nd ed. Englewood Cliffs/NJ: Prentice Hall, 1988.

[43] BURGHARDT J. Shows a simplified excerpt of the formal grammar for the C programming language, and a derivation of a piece of C code[EB/OL]. (2020). https://commons.wikimedia.org/wiki/File:C_grammar_stmt.pdf.

[44] CROCKER D, OVERELL P. Augmented BNF for Syntax Specifications: ABNF[EB/OL]. RFC Editor, 2008. https://www.rfc-editor.org/info/rfc5234. DOI:10.17487/RFC5234.

[45] NIELSEN H, MOGUL J, MASINTER L M, et al. Hypertext Transfer Protocol – HTTP/1.1[EB/OL]. RFC Editor, 1999. https://www.rfc-editor.org/info/rfc2616. DOI:10.17487/RFC2616.

[46] FROST R A, HAFIZ R. A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time[J/OL]. SIGPLAN Not., 2006, 41(5): 46-54. https://doi.org/10.1145/1149982.1149988. DOI:10.1145/1149982.1149988.

[47] DAVIES A. Async in C# 5.0[M]. " O'Reilly Media, Inc.", 2012.

[48] DAHL O. CAR Hoare Hierarchical Program Structures In: Structured Programming (Dahl, Dijkstra, Hoare)[M]. Academic Press, 1972.

[49] FOUNDATION P S. The Python Language Reference - Yield expressions[EB/OL]. (2019). https://docs.python.org/3/reference/expressions.html#yieldexpr.

[50] VASWANI A, SHAZEER N, PARMAR N, et al. Attention is all you need[J]. Advances in neural information processing systems, 2017, 30.